2016

Optimizing Virtual Machine I/O Performance in Cloud Environments

Tao Lu
Virginia Commonwealth University

OPTIMIZING VIRTUAL MACHINE I/O PERFORMANCE IN CLOUD ENVIRONMENTS

A Dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at Virginia Commonwealth University.

by

TAO LU

M.S., Huazhong Univ. of Sci. and Tech.: Sep. 2009 to Feb. 2012

B.S., Huazhong Univ. of Sci. and Tech.: Sep. 2005 to Jul. 2009

Director: Dr. Xubin He,

Professor, Department of Electrical and Computer Engineering

Virginia Commonwealth University

Richmond, Virginia

November, 2016

Acknowledgements

I thank my advisor, Prof. Xubin (Ben) He. I believed that Ben's recruitment to me was a miracle. I almost gave up my Ph.D. program application. It was Ben, who recruited me, trained me, inspired my potentials and made me an eligible Ph.D.. Ben is specially appreciated for his excellent academic tastes on judging my research motivations, clear logic on diagnosing my problem statements, and suggestive comments on my presentations.

I thank other committee members, Dr. Wei Cheng, Dr. Carl Elks, Dr. Qiqi Lu, and Dr. Weijun Xiao. They provided highly suggestive comments as early as I proposed my research plan. I thank their patience for enabling me to reschedule my proposal defense. I would also like to extend my thanks to other faculty and stuff members of the department of Electrical and Computer Engineering, in particularly to the administrative assistants Stacy E. Metz and Ellen Gresham.

I thank my STAR Lab mates, Morgan Stuart, Yuhua Guo, Kun Tang, Ping Huang, Pradeep Subedi. They provided me huge supports in the past four years. Morgan built the first virtual machine cluster in our lab which played a crucial role for my virtual machine migration research. Ping's family and my family had been neighbors for more than three years, we had wonderful time together such as visiting our mutual friends, sharing delicious foods, playing on the beach and carpooling. Ping's one-and-a-half-year-old son Daniel chased my two-and-a-half-year-old daughter Joy every day, which brought us a lot of fun. I also thank my department mates Yijie Huangfu and Qianbin Xia, with whom I have brotherhood.

I thank my friends Bob and Elaine Metcalf, Jim and Jan Fiorelli, Lex and Kate Strickland, Katie and Roman Dusil, Bud Whitehouse and Jennifer DeCerff. They have dedicated to serve Chinese students at VCU for five years. I got chances to

practice English with them. I received direct help and love from them. I thank Ed and Suping Boudreau, Geoffrey and Eunice Chan, with them I have joy and peace.

I thank my family, my wife Liping, my daughter Joy, my parents, brothers and sisters. My elder brother Dong taught and inspired me a lot before my middle school age. He guided and helped me all my way to the college. Liping dedicates herself to the family. She is doing a fabulous job in taking care of our daughter Joy. Every morning, Liping brews coffee, prepares breakfast, and packs my lunch. If my Ph.D. degree is an achievement, the better part of it should go to Liping. Joy is a joyful existence. Thanks, little guy.

Part of the research presented in this dissertation is sponsored by the US National Science Foundation under grants CNS-1320349 and CNS-1218960. I appreciate the sponsorship.

TABLE OF CONTENTS

LIST OF FIGURES

Abstract

OPTIMIZING VIRTUAL MACHINE I/O PERFORMANCE IN CLOUD ENVIRONMENTS

By Tao Lu

A Dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at Virginia Commonwealth University.

Virginia Commonwealth University, 2016.

Director: Dr. Xubin He,

Professor, Department of Electrical and Computer Engineering

Maintaining closeness between data sources and data consumers is crucial for workload I/O performance. In cloud environments, this kind of closeness can be violated by system administrative events and storage architecture barriers. VM migration events are frequent in cloud environments. VM migration changes VM runtime inter-connection or cache contexts, significantly degrading VM I/O performance. Virtualization is the backbone of cloud platforms. I/O virtualization adds additional hops to workload data access path, prolonging I/O latencies. I/O virtualization overheads cap the throughput of high-speed storage devices and imposes high CPU utilizations and energy consumptions to cloud infrastructures.

To maintain the closeness between data sources and workloads during VM migration, we propose *Clique*, an affinity-aware migration scheduling policy, to minimize the aggregate wide area communication traffic during storage migration in virtual cluster contexts. In host-side caching contexts, we propose *Successor* to recognize warm pages and prefetch them into caches of destination hosts before migration com-

pletion. To bypass the I/O virtualization barriers, we propose *VIP*, an adaptive I/O prefetching framework, which utilizes a virtual I/O front-end buffer for prefetching so as to avoid the on-demand involvement of I/O virtualization stacks and accelerate the I/O response.

Analysis on the traffic trace of a virtual cluster containing 68 VMs demonstrates that *Clique* can reduce inter-cloud traffic by up to 40%. Tests of MPI Reduce_scatter benchmark show that *Clique* can keep VM performance during migration up to 75% of the non-migration scenario, which is more than 3 times of the *Random* VM choosing policy. In host-side caching environments, *Successor* performs better than existing cache warm-up solutions and achieves zero VM-perceived cache warm-up time with low resource costs. At system level, we conducted comprehensive quantitative analysis on I/O virtualization overheads. Our trace replay based simulation demonstrates the effectiveness of VIP for data prefetching with ignorable additional cache resource costs.

CHAPTER 1

INTRODUCTION

Public cloud utilizes statistical multiplexing of computation, storage, and network to achieve elasticity, and the illusion of infinite resource capacity [3]. Virtualization techniques, which enable resource multiplexing, play a key role in data centers. Virtualization technologies bring the following noteworthy benefits. First, virtualization enables higher levels of resource utilization. Historically, in x86 server environments, companies would run just one application workload per server. The advent of server virtualization enables multiple workloads per server in virtualization environments. Second, virtualization techniques improve the manageability of data centers. The ability to migrate complete operating system instances without perceivable downtime, also known as live migration of virtual machine (VM), facilitates fault management, load balancing, system maintenance [20], network optimization [54], and cloud bursting [25]. Third, virtualization enables *software-defined* resource reservation, which provides huge flexibility to resource allocation in multi-tenancy environments.

A recent IBM research on production datacenters shows that the percentage of physical machines and VMs involving at least one occurrence of migration in a one-month observation window is 56.3% and 30.8%, respectively [12]. However, the benefits of VM mobility are gained with prices. VM migration brings overheads to applications from two aspects. First, mobility changes the network interconnection of VMs in a cluster. Analysis on Google Cluster Data shows that 96.2% of the VMs are not working alone. On average, a VM collaborates with 19.2 other VMs, and 49.1% of the VMs even have more than 1000 collaborator VMs. Thus, there is a high

chance that the migration of VMs will cause sub-optimal interconnection among VMs. Sub-optimal interconnection degrades the performance of a cluster. This problem is especially obvious in wide area migration scenarios. Second, mobility changes cache contexts of VMs. Cache contexts have important impact on VM performance. Since host-side warm pages are not transferred to the destination host during migration, a newly migrated VM suffers HDD-like performance until host-side cache is rebuilt. If the working set size of a VM is large (eg. Tens of GB), without a proper cache warm-up, the period of I/O performance degradation can be as long as hours.

Virtualization provides huge flexibility to resource allocation and infrastructure management in cloud environments. Unfortunately, I/O virtualization itself incurs additional overheads. Our tests on KVM virtualization platforms show that the virtual I/O sub-path adds an additional latency of about 60 μs. As high-performance NVM devices such as phase change memory (PCM) emerge, which delivers page I/O latency of about 1 μs, the overheads of virtual I/O become salient.

In this dissertation, we optimize VM storage performance in cloud environments. The optimizations are conducted from two aspects. First, we propose *Clique* and *Successor* mechanisms to mitigate the storage performance degradation caused by VM migration. Second, we propose *VIP* framework. Considering the virtual I/O sub-path has relatively long RTT but high bandwidth, *VIP* aims to avoid the virtual I/O bottleneck via front-end I/O merging and prefetching.

1.1 Affinity Grouping of Virtual Machines for Inter-Cloud Live Migration

Public cloud utilizes statistical multiplexing of computation, storage, and network to achieve elasticity, and the illusion of infinite resource capacity [3]. Virtualization techniques, which enable resource multiplexing, play a key role in data centers. The noteworthy benefits of virtualization techniques are twofold. First, virtualization

enables higher levels of resource utilization. Second, virtualization techniques improve the manageability of data centers. The ability to migrate complete operating system instances without perceivable downtime, also known as live migration of virtual machine (VM), facilitates fault management, load balancing, system maintenance [20], network optimization [54], and cloud bursting [25].

When VM migration happens in a single data center, where shared storage is available, storage migration can be avoided, just the CPU and memory states need to be migrated. As a result, the total migration time is usually less than 30s when the used memory size of the VM is less than 250 MB and without memory stress [29]. Even when the VM memory size is configured as 800 MB with memory stress, the memory migration costs less than 100s [20]. Storage migration, on the other hand, may take up to 1000s with a configured virtual disk of 40 GB in LAN environments with Gbps links.

Resource requirement of a cloud is fluctuant. To meet the QoS of peak load, resource is usually over-provisioned, which causes the resource underutilization when the load is low. The hybrid cloud model, which enables overloaded applications to be migrated to the public cloud [25], shows promise in solving this dilemma. The inter-cloud VM migration has two notable differences with the traditional intra-cloud VM migration. First, the inter-cloud link bandwidth is limited, due to the geographically distributed characteristic of data centers. Second, the storage of VMs needs to be migrated due to the inaccessibility of shared storage.

Cloud applications are usually complicated, and require the collaboration of a cluster of VMs. Analysis of Google Cluster Data [72] shows that a single job is served by 19.2 VMs on average, and can grow up to 4880 for large jobs. For a job consisting of several tasks, communication among tasks is common. Two typical examples are MapReduce [24] and MPI [32] applications. When performing VM migration in

cloud environments, existing VM selection policies, such as minimum migration time (MMT), random choice (RC), and the maximum correlation (MC) [9] all fail to consider the affinity of VMs during migration. If the VMs collaborating on a single job are split in geographically distributed clouds, the limited inter-cloud link bandwidth will dramatically degrade the application performance. This type of performance degradation was also reported in Pacer [78] and COMMA [77]. A potential strategy is to optimize the VM selection during migration to mitigate the impact of the network bottleneck to application performance. For example, if all of the VMs collaborating on an application are migrated in parallel, and resumed running at the destination simultaneously, then the VM communication through WAN can be avoided. However, if an application contains a large number of VMs, the migration can't converge [77]. For the aggregate block dirty rate of the VMs overwhelms the inter-cloud link bandwidth. As a result, grouping mechanisms are needed to partition the VMs into subgroups. Subgroups can then be migrated one at a time.

Analysis of a traffic trace of 68 VMs in an IBM production cluster [54] shows that the inter-VM traffic rate varies significantly. It implies that a judicious grouping strategy benefits the application performance during migration. Migrating a group of VMs with intensive traffic in parallel can avoid these VMs being split at geographically distributed locations during migration. We define the traffic-aware VM grouping problem as an optimization problem, and we propose a method called *Clique Migration*. Two implementations of *Clique Migration*, *R-Min-cut* and *Kmeans-SF* algorithms, are proposed. Inputs to the algorithms are the VM level traffic matrix, which contains the VM-to-VM traffic rate, and parameters indicate the number of subgroups. The algorithms output the policies indicate which VMs should be migrated together as subgroups, and the order in which each subgroup should be migrated.

1.2 Proactive Cache Warm-up of Destination Hosts in VM Migration Contexts

Shared storage is widely deployed as an IaaS cloud building block. In shared storage such as Amazon Elastic Block Storage (EBS), disks are located across the network [61]. Compared to CPU computation and DRAM access, the latencies of disk access and network transfer are relatively high. In IaaS clouds, multiple tenants access the storage components simultaneously. Concurrent access incurs network and disk contention. Therefore, as it has been observed in [2], often not the computing frontend, but the storage components, the disk backend, and the network fabric, are the performance bottlenecks in IaaS clouds.

Caches are employed to save remote data locally so as to shorten the data access path and reduce access latency. For example, FS-Cache [28], a network filesystem caching facility, trades client-side local storage space to gain performance improvements for access to slow IP networks. The web caching [33] is also widely deployed to accelerate page response of web browsers. In *Bing* web search, each worker server uses tens of GB of DRAM for caching to reduce the average amount of disk I/O to less than 0.3 KB/s per server [26].

In cloud environments, I/O requests from applications traverse VMs, hosts, networks, and finally complete on storage servers. In virtualization systems, host-side caches are critical for improving VM storage performance. Many optimizations of host-side caches have been proposed. To achieve better cache efficiency, eviction-based page placement policies [35, 48] and content-based page deduplication [62] have been implemented to make the VM direct cache and the host-side cache be exclusive. Cache pooling [31], and SSD-based caching [17, 15, 70, 71, 52, 4, 53] have been implemented to increase the cache efficiency or capacity. These existing works

bring host-side caching to the stage.

VM storage performance benefits brought by warm host-side caches are voided by VM migration, a common activity in today's datacenters. The live migration of VMs facilitates fault management, load balancing, system maintenance [20], datacenter network optimization [54], and cloud bursting [25], etc.. A recent IBM research on production datacenters shows that the percentage of physical machines and VMs involving at least one occurrence of migration in a one-month observation window is 56.3% and 30.8%, respectively [12]. Since warm pages in the source host are not transferred to the destination host during migration, a newly migrated VM suffers HDD-like performance until the cache is rebuilt. If the working set size of the VM is large (eg. Tens of GB), without a proper cache warm-up, the period of I/O performance degradation can be as long as several hours [4]. This performance degradation has been reported in [15, 22, 10].

There are three potential methods to mitigate the VM I/O performance degradation due to the cache coldness after VM migration. First, a shared cache pool [31] can be deployed to enable post-migration accessibility of cached pages. Second, pages residing in host-side caches can be migrated during migration process [70]. Third, existing storage-side cache warm-up mechanism such as Bonfire [75] can be customized to preload data into caches after VMs resume running on destination hosts. However, all these methods have limitations. The first method is restricted to local areas since low-speed wide area networks limit the benefits of cache pooling. The second method may considerably prolong the total VM migration time if the cache footprint to be migrated is large. The third method is practical, but during the cache warm-up period, VMs will undergo extreme I/O performance degradation due to the I/O contention caused by aggressive data preloading.

We propose *Successor*, which proactively warms up caches of destination hosts

before migration completes. *Successor* ensures that post-migration VM storage performance is as good as the pre-migration performance. Specifically, accessibility of destination hosts during migration enables *Successor* to parallelize cache warm-up and VM migration. Being different from storage-side cache warm-up [75] that only a single machine is involved, VM migration involves two active physical machines, which enable running VM on the source host and concurrently conducting cache warm-up on the destination host. Parallelizing the cache warm-up and VM migration brings challenges. In the memory-only migration scenario, there is a *write-after-prefetch* cache consistency problem. That's if a page is modified in the source host after it has been prefetched into the cache of the destination host, the prefetched page becomes stale. In storage migration scenario, there is a warm-up I/O path problem. That's VM disk images still reside in source hosts during VM migration, destination hosts are not able to build local cache footprints of migrating VMs before migration completes. We propose *dirty page tracking* and *piggyback warm-up on migration* mechanisms to address these two problems, respectively. We implement *Successor* on a QEMU/KVM [8, 39] virtualization platform. Tests show that parallelizing cache warm-up and VM migration can be achieved with *Successor*.

1.3 System-level Optimization of Virtual I/O

VM-side caches can obviate the I/O virtualization overheads so as to improve the VM storage performance and system energy efficiency. More specifically, based on our tests for 4KB read requests at the IOPS of 5k, hypervisor-side DRAM caches consume about 3x the power and delivers 30x the per I/O latency of VM-side DRAM caches. One of our servers hosts four I/O intensive VMs, the hypervisor-side caching consumes 48 watts while the VM-side caching consumes only 13 watts. The idle power of the server is 121 watts. In other words, comparing with hypervisor-side caching,

VM-side caching can save about 25% of the entire server's active power.

Although the performance and energy efficiency of VM-side caching are much higher than hypervisor-side caching, building a large VM-side DRAM cache is costly. For example, doubling the DRAM size of a Google *n1-standard-1* VM instance increases the whole VM's price by 40%. From the viewpoint of system management, larger VM DRAM allocation means lower VM density, which is the number of VMs can be supported by a physical machine, since the DRAM capacity is a key limitation factor of VM density. Therefore, it's not cost-efficient to simply deploy a large VM-side DRAM cache for performance and energy efficiency purpose.

Fortunately, the transfer bandwidth between the virtual I/O front-end and the back-end is high, that provides a chance to implement front-end prefetching so as to avoid the I/O virtualization bottleneck. Specifically, for application running inside the VM, our tests show that for 4KB small I/O requests, the maximum throughput of hypervisor-side caching is only about 3% of the VM- side caching. For 1MB large I/O requests, the maximum throughput of hypervisor-side caching is nearly the same as the VM-side caching. Therefore, if a batch of small front-end requests can be merged as a large request, the I/O virtualization overhead can be amortized. As a result, VM applications perceive improved throughput and the system energy consumption is also reduced. The I/O batch submission has also been employed to improve the throughput of multi-queue SSDs in I/O virtualization platforms [37]. However, as it's stated in [37] that if the polling interval of the batch submission is too short, the throughput gain is very limited. Simply increasing the polling interval will considerably increase the latency of part of the requests.

Prefetching can obtain the similar benefits of I/O batch submission, but avoid the increased I/O response time caused by long polling intervals. Instead of batching on-demand requests, if correlated blocks can be predicted, batched and prefetched into

the VM-side cache, future read requests can be accelerated without a long queuing latency. Our tests on Google Compute Engine show that even the naive sequential front-end prefetching increases the VM random read I/O throughput by 2.7x and shortens the response time of more than 95% of the requests.

An optimal prefetching needs to take the block correlation as well as the performance characteristics of the storage devices into consideration. Since not all workloads have strong spatial locality, a native sequential prefetching may read considerable unused data into the cache. Prefetching is not free. Prefetching unused data is a waste of storage device bandwidth and cache space. Also, the prefetching process competes system resources with the user-oriented applications, and degrades the application performance.

We propose *VIP*, which utilized Markov chains to recognize I/O correlation sequences which in turn are used for prefetching. VIP makes the following contributions. First, we extensively evaluate the impact of virtual I/O on VM storage performance and on system energy consumption. Second, we analyze a group of block I/O traces and quantify workload features to validate the potential gains of virtual I/O front-end prefetching. Third, we propose *VIP*, which adaptively preloads data from the back-end cache into the front-end cache so as to improve the VM performance and the system energy efficiency. Through block I/O trace based simulations, we verify the effectiveness of VIP algorithm. *VIP* enables tenants of public clouds to optimize storage performance without the involvement of cloud service providers. Also, IaaS providers can integrate *VIP* into their VM instance templates to improve virtual I/O performance in a tenant-transparent way. *VIP* can be stacked above virtual I/O optimizations such as *VIO-prefetching* [19] and Virtio-blk Multi-queue [43, 38].

1.4 Dissertation Organization

The remainder of this dissertation is structured as follows. In Chapter 2, we introduce *Clique Migration*, a migration optimization mechanism that exploits the affinity of VMs to maximize cluster performance during its inter-cloud live migration. Based on Clique Migration, we propose and implement two algorithms called R-Min-Cut and Kmeans-SF. Analysis of the traffic trace of 68 VMs in an IBM production cluster shows that our algorithms can reduce inter-cloud traffic by 25% to 60%, when the degree of parallel migration is from 2 to 32. Tests of MPI multi-PingPing benchmark running on simulated inter-cloud environments, show that our algorithms can significantly shorten the period during which applications undergo performance degradation. Tests of MPI Reduce scatter benchmark show that R-Min-Cut can keep the performance during migration at 26% to 75% of the non-migration scenario.

In Chapter 3, we present *Successor*, a proactive cache warm-up mechanism for destination hosts in virtual machine migration contexts. Based on the observation that destination hosts are active during migration, Successor parallelizes destination cache warm-up and VM migration to proactively prefetch warm pages into destination caches before migration completion, so that the cache contexts as well as the post-migration performance of VMs can be maintained without degradation. Compared with *migrating host-side cache* and *Bonfire*, *Successor* achieves zero VM-perceived cache warm-up time with low resource costs and performance penalties.

In Chapter 4, we present *VIP*, an adaptive virtual I/O front-end prefetching mechanism. VIP prefetches correlated data from the back-end (hypervisor-side) cache into the front-end (VM-side) cache when a back-end access is inevitable, so as to take the performance benefits of VM-side caches without losing the capacity benefits of hypervisor-side caches. Compared with hypervisor-side caching, VIP achieves faster

virtual I/O response and lower system energy consumption with a cost of limited additional VM-side DRAM resource.

In Chapter 5, we discuss research efforts related to this dissertation. In Chapter 6, we summarize our work and discuss various directions for future work motivated by the work of this dissertation.

CHAPTER 2

AFFINITY GROUPING OF VIRTUAL MACHINES FOR INTER-CLOUD LIVE MIGRATION

Affinity is common among Virtual Machines (VMs) in cloud environments. If VMs collaborating on a job are split in geographically distributed clouds, the low bandwidth and high latency inter-cloud communication via a wide area network (WAN) will dramatically degrade the system performance. A potential solution is migrating all of the VMs collaborating on a job in parallel, so as to avoid wide area communication. However, if the job is too large, it becomes impractical to migrate all of the VMs simultaneously due to limited WAN bandwidth and high block dirty rate. We propose a migration optimization mechanism called *Clique Migration* to partition a large group of VMs into subgroups based on the traffic affinities among VMs. Then, subgroups are migrated one at a time. Based on *Clique Migration*, we propose and implement two algorithms called *R-Min-Cut* and *Kmeans-SF*. Analysis of the traffic trace of 68 VMs in an IBM production cluster shows that our algorithms can reduce inter-cloud traffic by 25% to 60%, when the degree of parallel migration is from 2 to 32. Tests of MPI multi-PingPing benchmark running on simulated inter-cloud environments, show that our algorithms can significantly shorten the period during which applications undergo performance degradation. Tests of MPI Reduce_scatter benchmark show that *R-Min-Cut* can keep the performance during migration at 26% to 75% of the non-migration scenario.

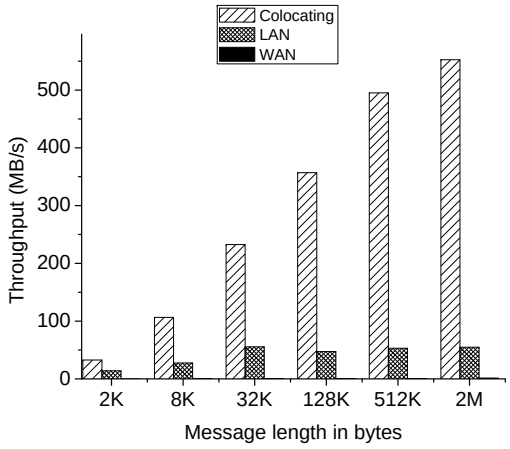


Fig. 1. The Throughputs of Intel MPI Multi-PingPing benchmark using different network configuration.

2.1 Problem Description

We consider a scenario where multiple VMs need to be migrated from one data center to another. In this scenario, we seek to mitigate the performance degradation of applications during the live migration of VMs. We name our migration optimization mechanism *Clique Migration*.

Clique Migration is proposed based on several observations: (1) Network bandwidth is often the bottleneck of cluster systems, especially when the applications are I/O intensive. (2) In cloud environments, VMs are not working alone. (3) When the available migration bandwidth for a VM is limited, and the block dirty rate of the VM is relatively high, the migration fails to complete. (4) The traffic affinities among VMs are different.

Network communication has long been the bottleneck of cluster systems for scientific computation [65]. For modern cloud applications, a sudden explosion of network traffic can significantly deteriorate the application performance [58]. Even when the network is not saturated, remote access can still cause significant performance degra-

dation [67]. As a result, when the interactive components of MPI, batch processing, or multi-tier applications are split in geographically distributed data centers, the low bandwidth, high latency link between data centers will dramatically degrade the application performance. This observation is also reported in Pacer [78], COMMA [77], and M. Cardosa's work [16].

Figure 1 shows our tests with Intel MPI1 Multi-PingPing benchmark using different network configurations on a cluster, which contains 8 VMs hosted on 4 physical machines. Tests show that when a VM pair are collocated at the same host, the VM-to-VM throughput achieves 552.6 MB/s. When the VMs are interconnected via 1 Gbps LAN, the throughput achieves 54.8 MB/s on average. However, the throughput drops to less than 1.2 MB/s when we limit the link bandwidth to 100 Mbps to simulate WAN. This group of tests show that the link bandwidth has an obvious impact on application performance, especially when the applications are communication intensive.

The performance of inter-cloud communication decides the performance of geographically distributed applications. We use inter-cloud communication volume as the metric to measure the performance degradation of applications during migration. The larger the communication volume during migration, the more application performance will be degraded. The communication volume is the product of the traffic rate and communication time. The communication time depends on the migration time, as well as the order in which each subgroup of VMs are migrated. This is because in the wide area migration scenario, the longer migration takes, the more opportunity it gives the VMs to generate dirty data at the source site. Intuitively, if all the VMs collaborating on a job can be migrated in parallel, and resumed running at the destination data center simultaneously, then the inter-cloud VM-to-VM communication as well as the application performance degradation can be avoided.

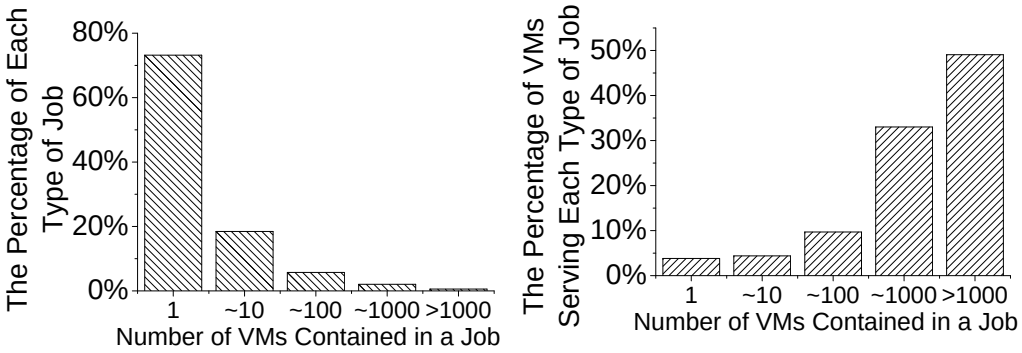


Fig. 2. The collaborative behavior of VMs in Google workloads.

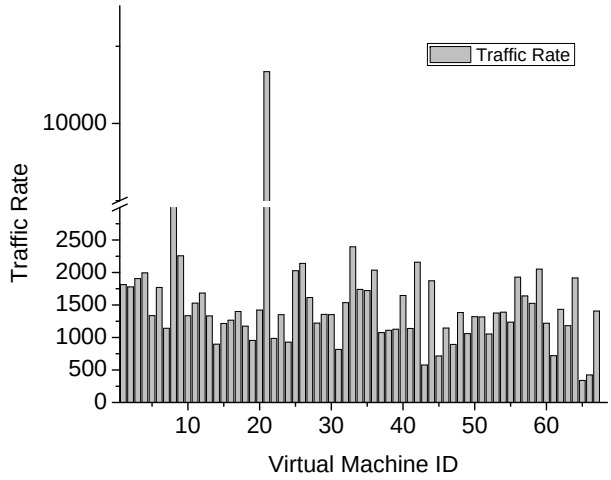


Fig. 3. The traffic rate of each VM to all the other VMs.

However, based on observation 2, the number of VMs collaborating on a job can be up to several thousands. Figure 2 demonstrates our analysis of Google Cluster Data [72]. We define "sibling" as the VMs collaborating on a single job. The trace contains more than 3 millions observations, 9218 unique jobs and 176,580 unique tasks. The analysis shows that 96.2% of the VMs in the cloud are not working alone. On average, a VM has 19.2 siblings, and 49.1% of the VMs have more than 1000 siblings. As a result, if all of the VMs serving a single job are migrated simultaneously, the migration bandwidth allocated to each VM is limited. Bradford's work [13] has shown

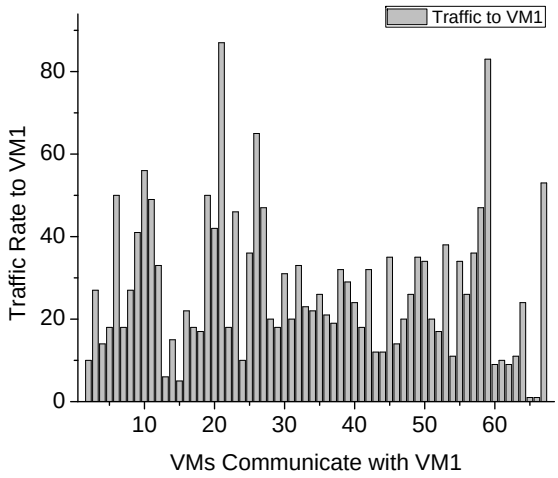


Fig. 4. The traffic rate between VM1 and each of the other VMs.

that if the VM to be migrated writes to the disk at a very fast rate, it generates a lot of dirty data which prevents the migration from progressing [13]. The similar observation of migrating multiple VMs in parallel was also reported in COMMA [77]. The I/O throttling mechanism proposed in [13] can be used to enforce migration convergence, but it will degrade the application performance dramatically. To mitigate performance loss, VMs need to be partitioned into several subgroups based on traffic affinity, with each subgroup migrated as an ensemble in parallel.

As the information about the inter-VM traffic rate is not available in Google Cluster Data [72], we use another trace from an IBM production cluster system [54] to show the traffic affinities among VMs. The trace contains the network connection rates of 68 VMs. The connection rate is the number of TCP connections in unit time. We use this connection rate as traffic rate in our paper, similar to its use in the original literature [54]. The inter-VM traffic rate can be found in Meng's paper [54]. Here we simply show each VM's total traffic rate, and traffic rates between VM1 and all the other VMs. From Figure 3, we can see that each VM's total traffic rate varies. Figure 4 shows that different VM pairs have significantly different traffic intensities.

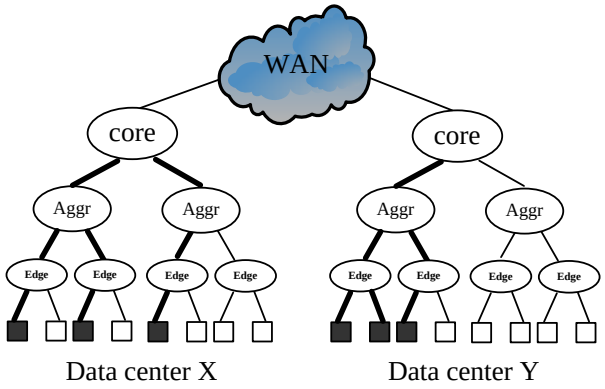
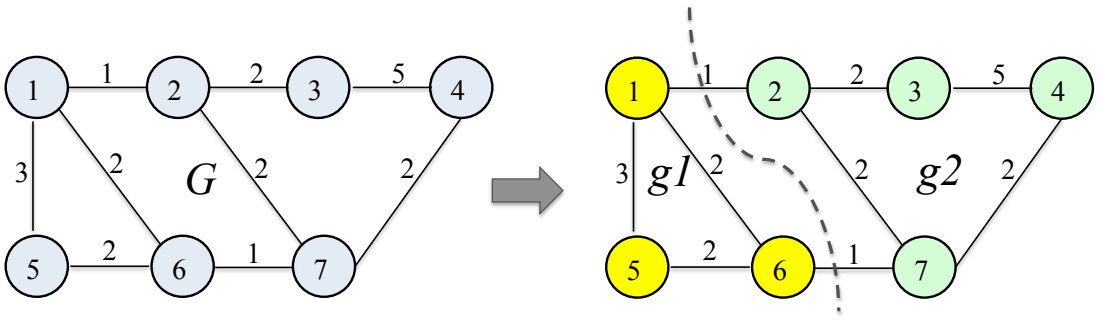


Fig. 5. Overview of the *Clique Migration*.

Observation 4 implies that different grouping strategies have different impacts on the inter-cloud communication, which affects the application performance. Meng's paper [54] has shown that per-VM traffic is relatively constant at a large time scale. This observation implies the predictability of VM traffic, further motivating the practicality of *Clique Migration*.

The overview of *Clique Migration* is illustrated in Figure 5. We use a simplified tree-like switch interconnection to demonstrate the inter-cloud VM migration scenario. We assume a predefined group of VMs G needs to be migrated from data center X to data center Y. We assume X and Y are geographically distributed, and interconnected via WAN. As a result, the virtual disk must be migrated. The migration is performed in the *pre-copy* [39] manner, in which the storage is migrated first, then the memory and CPU states.

When the migration is initiated, all of the VMs are located at data center X. We assume the number of VMs contained in G is too large to be migrated all together simultaneously. G needs to be partitioned into n subgroups. Each subgroup is denoted as g_i $(1 \leq i \leq n)$. When migrating g_i $(i \geq 2)$, the subgroup sets $\cup_{j=1}^{i-1} g_j$ and $\cup_{k=i}^{n} g_k$ are split in data centers X and Y. During the migration of g_i, the total



a. VM Cluster *G* Contains 7 VMs

b. *G* is partitioned into *g1* and *g2* via *min-cut*

Fig. 6. Cluster G is partitioned into *g1, g2* based on *min-cut algorithm.*

wide area communication volume V_i is denoted as $R_{\cup_{j=1}^{i-1} g_j, \cup_{k=i}^{n} g_k} T_{g_i}$. As a result, during the whole migration process, the inflicted wide area communication volume V is $\sum_{i=2}^{n} R_{\cup_{j=1}^{i-1} g_j, \cup_{k=i}^{n} g_k} T_{g_i}$.

As we use inter-cloud communication volume as the metric to measure the performance degradation, we aim to minimize V. V is determined by the inter-subgroup traffic rate, the migration latency of each subgroup, and the order in which each subgroup is migrated. We individually discuss each of these subproblems in the following section.

2.2 Modeling and Analysis

To minimize inter-cloud communication volume, we propose grouping mechanisms to determine which VMs should be migrated together as subgroups, and a shuffling mechanism to decide the order in which each subgroup should be migrated. Specifically, we propose and implement two independent algorithms *R-Min-Cut*, and *Kmeans-SF* as potential solutions.

2.2.1 Grouping Mechanism

Assume we have a group of virtual machines G that needs to be migrated. Graph a of Figure 6 shows the traffic graph of G. Each node represents a VM, and each edge represents the traffic rate between each VM pair. If VMs in G are too numerous to be migrated all together, G has to be partitioned into several subgroups, and each VM in a subgroup can be migrated in parallel. To minimize wide area communication volume during migration, the traffic rate between subgroups $g1$ and $g2$ should be kept as low as possible.

The central aspect of the grouping mechanism is a graph partitioning algorithm. The input to the algorithm is a traffic graph, which contains the VM-to-VM traffic rate. The outputs are two subgraphs with minimum inter-subgroup traffic rate. Therefore, the grouping problem can be transformed to classic *min-cut* problem. The min-cut algorithm [66] can find the minimum cut of an undirected, edge-weighted graph. The procedure of the algorithm can be summarized as follows. Suppose an ordinary undirected graph G represents the traffic flow among all the VMs. The vertex set V represents the VMs, and the edge set E represents the traffic relation among VMs. Every edge e has a nonnegative real weight *w(e)*, which specifies the quantitative traffic rate between VM pairs. Let s and t be two vertices of G, and $G/\{s,t\}$ be the graph evolved from merging s and t. Then, a minimum cut of G is the smaller one of a *minimum s-t-cut* of G and a minimum cut of $G/\{s,\ t\}$. So a *minimum s-t-cut algorithm* can be used to construct a recursive algorithm to find a minimum cut of a graph. The detail of the algorithm can be found in [66]. An example in Figure 6 b demonstrates what the *min-cut* algorithm is expected to achieve. The expected partitions of cluster G are $g1$=$\{1,5,6\}$, $g2$=$\{2,3,4,7\}$ with weight w= 2.

Algorithm: `R-Min-Cut`

Input: An ordinary undirected graph G with vertex set V and edge set E, edge weight set W contains nonnegative real weight $w(e)$ of every edge e, an arbitrarily selected vertex a, size of each subgroup $S_1, ..., S_k$.

Return: Sub-groups.

Procedure:

$\{G_1, G_2\}$=MinimumCut (G, W, a); /* Compute the binary min-cut of G using Stoer-Wanger MinimumCut algorithm; */

Sort $\{G_i\}$ by decreasing size;

index=1;

for i=1 to 2 **do**

subsize=$|G_i|$;

for i=1 to subsize **do**

pick a VM v which incurs minimal WAN traffic from G_i;

ordered[index]=v;

index++;

delete v from G_i;

for m=1 to k **do**

pick S_m elements serially from array *ordered*, and insert these elements into subgroup g_m;

return k subgroups $\{g_1, ..., g_k\}$ each with a size of $S_1, ..., S_k$

For a binary partition, *min-cut* is theoretically an optimal solution. However, binary partition may be not enough for partitioning a group of VMs in a practical scenario, because each subgroup can still be too large for migration. A further min-cut on the subgroups seems to be a feasible solution. However, a further *min-cut*

partition can't guarantee that the wide area traffic remains minimal. That's to say, if $g2$ is still too large to be migrated, it can be further partitioned into $g2_a$, and $g2_b$ based on *min-cut*. This partition can guarantee that $R_{g2_a,g2_b}$ is minimal. However, it can't guarantee that $R_{g1 \cup g2_a,g2_b}$ is minimal. We propose a greedy algorithm called *R-Min-Cut*, to recursively maintain a minimal wide area traffic rate during migration. In the above described scenario, instead of performing a further *min-cut* on $g2$, *R-Min-Cut* will select VMs one by one from $g2$ to decide the order in which VMs in $g2$ should be migrated. In each selection, *R-Min-Cut* uses an exhaustive comparison to guarantee that the wide area traffic rate is minimal. The size of each subgroup S_m $(1 \leq m \leq k)$ can be predefined, or decided dynamically based on the block dirty rate and available migration bandwidth. The pseudo code of *R-Min-Cut* is described in Algorithm R-Min-Cut.

k-means clustering [47] is a method of vector quantization that's popular for cluster analysis in data mining. k-means aims to partition n observations into k clusters, in which each observation belongs to the cluster with nearest mean. As the traffic rate among all the VMs can be presented as a traffic matrix, each row of the matrix is a natural traffic vector of a VM. We also explore the feasibility of k-means to minimize inter-subgroup traffic. Although k-means is computationally difficult, there are efficient heuristic algorithms that are commonly used and converge quickly to a local optimum.

Algorithm: `Kmeans-SF`

Input: A traffic matrix T contains VM-to-VM traffic rate, parameter k specifies the number of result subgroups.

Procedure:

G_T=`kmeans(T, k);` /* Compute the k-means clusterings of T using Kendall tau similarity metric; */

`Shuffle(`G_T`)`

Algorithm: `Shuffle`

Input: G_T, the result subgroups of kmeans clustering on T.

Return: Shuffled sub-groups.

Procedure:

`Find a` g_i `from` G_T `such that` $\frac{R_{g_i,G\setminus g_i}}{L_{g_i}}$ `is minimal;`

`shuffled[1]=`g_i`;`

`Delete` g_i `from` G_T`;`

for i=2 to k **do**

`Find a` g_i `from` G_T `such that` $\frac{R_{g_i,shuffled}}{L_{g_i}}$ `is maximal;`

`shuffled[i]=`g_i`;`

`Delete` g_i `from` G_T`;`

return `shuffled;`

2.2.2 Shuffling Mechanism

After determining which VMs will be grouped together via grouping mechanism, the order in which subgroups are migrated has considerable impact on the application

performance. For example, assume a group of VMs G is partitioned into $g1$ and $g2$, which consist of 2 and 5 VMs respectively. Also we assume the migration time of 2 VMs is $2T$, and the migration time of 5 VMs is $5T$. Let R represent the traffic rate. If $g1$ is migrated first, followed by $g2$, then the total wide area traffic during migration is $5T \times R_{g1,g2}$. On the other hand, if $g2$ is migrated first, followed by $g1$, then the total wide area traffic during migration is $2T \times R_{g1,g2}$. From this analysis, we can see that even for the same grouping decision, different orders in which subgroups are migrated considerably affects the amount of wide area traffic during migration.

To decide the order in which subgroups should be migrated, we have to answer two questions. First, which group should be migrated first when migration is initiated? Second, based on the already migrated groups, which group should be migrated subsequently? When migrating the first subgroup, it will still be running at the source site, thus, there is no wide area communication during this step. After the completion of this step, the migrated subgroup of VMs will communicate with the VMs located at the source site, and the wide area communication occurs. Based on the analysis, if the first subgroup is larger, it will stay with the other VMs at the source site longer, and less wide area communication occurs. Also, if the first subgroup is more isolated, it will cause less wide area communication. As a result, we use $\frac{R_{g_i,G\setminus g_i}}{L_{g_i}}$ as criteria for picking the first subgroup to be migrated. From all of the subgroups, the one selected to be migrated should have the minimum quotient of traffic rate and migration latency. After the migration of the first subgroup completes, the following subgroup should be picked to immediately reduce wide area traffic rate. Thus, the subgroup with higher traffic with the already migrated VMs, and with shorter migration latency, will be picked as the migration candidate.

For *R-Min-Cut* algorithm, shuffling is not required. Because *R-Min-Cut* employs a greedy strategy which implies a chronological order. But for k-means, the clustering

Table 1. A List of Symbols

Symbols	Description
D	Disk size of a VM
D_{dirty}	Dirty data size during the last iteration of migration
W	Total WAN bandwidth
R	Block dirty rate of a VM
L_i	Latency of the i^{th} iteration of migrating a VM
L	Total latency of migrating a VM
N	Number of VMs in parallel migration
D_i	Disk size of the i^{th} VM in parallel migration
W_i	Bandwidth allocated to the i^{th} VM in parallel migration
R_i	Block dirty rate of the i^{th} VM in parallel migration
L_i^p	Latency of migrating the i^{th} VM in parallel migration
L^p	Total latency of migrating all N VMs in parallel

is a static process, a shuffling process is needed. With shuffling mechanism, an optimized k-means algorithm called *Kmeans-SF* is described in Algorithm Kmeans-SF.

2.2.3 Modeling the Migration Latency

Migration latency is an important parameter for our shuffling mechanism. Because migration latency decides the period during which applications undergo performance degradation. A model to accurately predict the VM migration latency is needed. Related work have been discussed in [76, 78, 46]. A list of symbols used in this section are summarized in Table 1.

We use KVM [39] as the VM hypervisor in our testbed. The storage migration of KVM employs the *pre-copy* migration model. Therefore, all of our analysis related to storage migration in this paper assumes *pre-copy*. That's virtual disk will be migrated prior to memory copying. Storage migration of KVM employs a dirty block tracking (DBT) mechanism. With the DBT mechanism, the virtual disk will be migrated from beginning to end during the first iteration, and if a block has been modified after it has been copied during migration, it will be tracked and copied again. In practice, KVM employs a dirty block log, which provides a bitmap of modified blocks since

the last migration. The storage migration is an iterative process, when the amount of blocks remaining is reduced to a threshold, the memory migration begins. The memory migration performs in a similar manner to storage migration. When the number of dirty pages reduces to a threshold, the system will be shutdown, and the left-over dirty blocks, pages, and CPU state will be copied to the destination. The VM then resumes running at the destination.

The disk size is usually about two orders of magnitude of the memory size, which makes the storage migration our primary consideration. Suppose a VM with virtual disk size of D, migration speed of W, and block dirty rate of R. The latency of migrating this VM can be calculated as follows.

First of all, the disk is migrated from the beginning to the end. The migration latency of the first round iteration is calculated according to Equation 2.1.

$$L_1 = \frac{D}{W} \tag{2.1}$$

During the migration's first round, the size of dirty blocks accumulated can be calculated according to Equation 2.2.

$$D_{dirty} = \frac{RD}{W} \tag{2.2}$$

Thus, the latency of second round migration is calculated according to Equation 2.3.

$$L_2 = \frac{D_{dirty}}{W} = \frac{RD}{W^2} \tag{2.3}$$

The latencies of all rounds compose a geometric progression with the common ratio of $\frac{R}{W}$. The total latency of migrating the VM can be calculated according to Equation 2.4.

$$L = \frac{D}{W - R} \tag{2.4}$$

For parallel migration, suppose N VMs are migrated simultaneously, the latency of migrating the i^{th} VM can be calculated according to Equation 2.4. The latency is shown as Equation 2.5.

$$L_i^p = \frac{D_i}{W_i - R_i} \tag{2.5}$$

Finally, the total latency of migrating all the N VMs is shown as Equation 2.6.

$$L^p = \max_{1 \leq i \leq N} \frac{D_i}{W_i - R_i} \tag{2.6}$$

2.3 Evaluation

To verify the effectiveness of *Clique Migration*, we perform trace based analysis, as well as tests on a practical prototype system. For trace based analysis, we use the inter-cloud communication volume during migration as the metric to measure the performance of both *R-Min-cut* and *Kmeans-SF* algorithms. For tests on simulated inter-cloud environments, we directly measure the performance of MPI benchmarks in the testbed.

We run Intel MPI benchmarks 4.0 beta [32] in our prototype KVM-based virtual machine cluster. Specifically, in one scenario, we run parallel transfer benchmark

Multi-PingPing on 8 VMs to measure the performance of *Clique Migration* on a group of VMs with obvious Cliques. In another scenario, we run a collective benchmark *Reduce_scatter* on 32 VMs to measure the performance of *Clique Migration* in an environment with more complex inter-VM traffic affinity.

The experiment setup is as follows. We use a data center interconnection structure described in Figure 5 as our reference model. We use a 24-port HP 1810-24G switch as the core switch. We use another two NETGEAR ProSAFE 8-Port GS108 as edge switches, which are connected to the core switch. Each edge switch connect 4 physical machines. Each physical machine has two 2.30GHz Intel(R) Xeon(R) CPU E5-2630 processors, 64GB RAM, 500 GB hard drives, CentOS with Linux kernel version 2.6.32, QEMU and KVM. All of the VMs located in physical machines are configured with 8 GB disks, 2 GB of RAM, and a single VCPU. Normally, the link bandwidth between edge switch and core switch is 1 Gbps. In the scenario of simulating WAN, we limit this link bandwidth to be 100 Mbps via configuring the core switch parameters.

2.3.1 Accuracy of The Latency Model

Migration latency is an important metric for making a migration decision. In the storage migration scenario, both the virtual disk and memory need to be migrated. As the virtual disk is usually one or two orders of magnitude larger than memory, the migration latency is dominated by the disk size. With a certain disk size, available migration bandwidth, as well as block dirty rate have direct impact on the migration latency [29].

To measure the accuracy of the migration latency model we conduct extensive tests on our platform. As the latency model of parallel migration is a direct extension of migrating a single VM, we focus on testing the migration latency of a single VM

Table 2. Evaluation of VM migration latency with different block dirty rates.

Parameters collected with different *fio* write rates (MB/s); (The units of W, R and *Measured Latency* are MB/s, MB/s and second;)					
	0	0.25	1	4	16
W	85.5	72.1	72.1	72.5	68.8
R	0	0.25	0.98	3.3	4.7
Measured Latency	93.4	110.8	112	113.6	120.1

with varying block dirty rates. The migration latency of parallel migration is decided by the last completed VM. For parallel migration, we focus on observing the difference in completion time for each VM.

We use *fio*, a standard Linux I/O tester, to generate disk pressure during

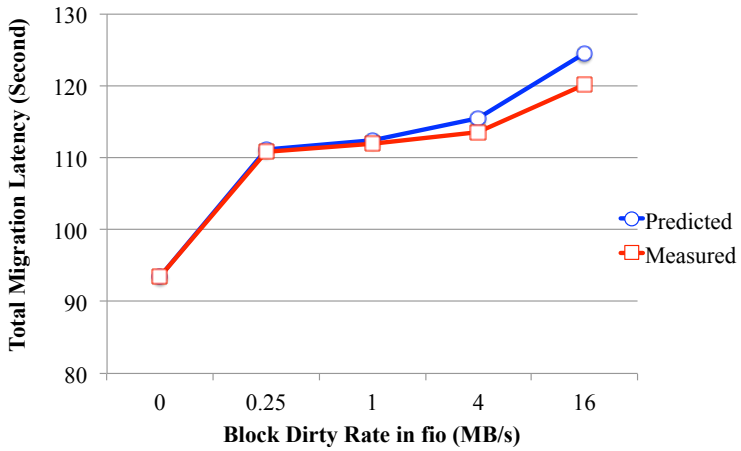


Fig. 7. Comparison of predicted and measured migration latency of a single VM with different block dirty rates in *fio*.

migration. We set the *fio* in *randwrite* and *sync* mode, with *size* of 256 MB. Meaning that *fio* will perform random write directly to the disk in the range of 256 MB. Also, we specify the write rate as 0.25, 1, 4, and 16 MB/s respectively, in the tests. The actual migration bandwidth W, as well as the actual block dirty rate R in each test are shown in Table 2. The measured base data D migrated during migration is 7989.6 MB. The comparison of predicted and measured latency of migrating a

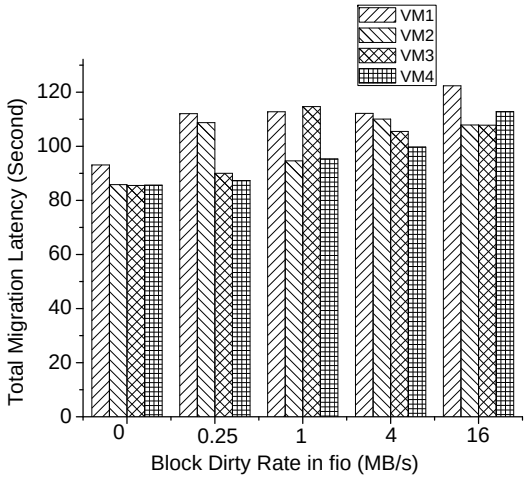


Fig. 8. Migration completion time of each VM in the scenarios of migrating 4 VMs in parallel with different block dirty rates in *fio*.

single VM with different block dirty rates is shown in Figure 7. The tests show that with configured block dirty rate ranges from 0 to 16 MB/s, the migration latency model has an error of less than 4%. Note that because of I/O interference, when *fio* executes, the migration speed has an obvious degradation. The reason is that the random write of *fio* affects the read performance of migration process. Also, when the specified random write of *fio* is larger than 4 MB/s, the specified speed cannot be achieved because of disk contention.

For parallel migration, we migrate 4 VMs simultaneously from 4 different physical machines. The migration is performed via infiniband. The migration latency is shown in Figure 8. The tests show that without a control mechanism, the 4 VMs can't complete migration at the exactly same time. The first migrated VM and the last migrated VM have migration latency difference of up to 20%. This gap will be much larger if the configured virtual disk size dramatically varies. In practical scenario, this latency gap will cause wide area traffic during migration. Therefore, a synchronization mechanism like Pacer [78] is necessary.

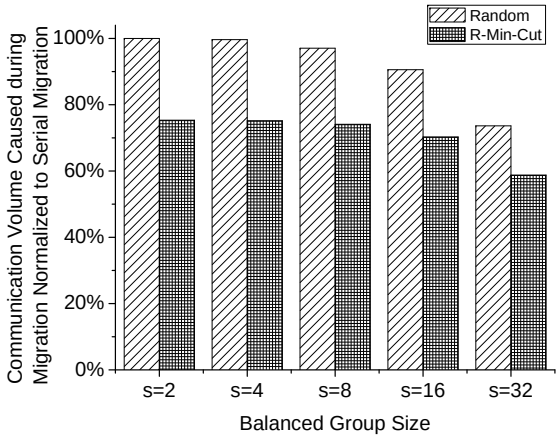


Fig. 9. Comparison of traffic savings with different balanced group sizes.

2.3.2 Parallel Migration vs. Serial Migration

The motivation of *Clique Migration* is that serial migration will split the collaborating applications in geographically distributed clouds, resulting in the degradation of application performance. In order to show the benefits of parallel migration, we analyze the communication volume during migration in serial and parallel migration procedures using IBM cluster trace [54]. The inter-cloud VM migration is a dynamic process, when the placement of VMs located at the source or at the destination change, the inter-cloud traffic rate changes. We analyze each state of the VM placement during migration, calculate the inter-cloud traffic rate in a certain state, as well as the duration of each state. We sum up the product of traffic rate and duration in each state as the total size of communication volume during migration.

For a more accurate comparison, we use balanced group sizes in parallel migration. We partition all 68 VMs using group sizes from 2 to 32 with an exponential increment. When the group size is 32, there are actually 3 groups with sizes of 32, 32 and 4. For a more comprehensive comparison, we use *Random* grouping, as well as

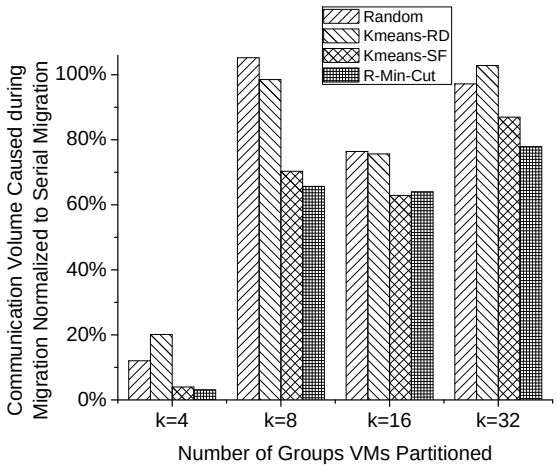


Fig. 10. Comparison of Traffic savings with different Grouping, and Shuffling Mechanisms.

R-Min-Cut grouping mechanism, to measure the benefits of parallel migration. The test results are shown in Figure 9. We use communication volume of serial migration during migration as the baseline. Tests show that even with a *Random* grouping mechanism which doesn't implement any optimization, parallel migration reduces communication volume by up to 26.4% compared to serial migration. However, when the group size is as small as 2 or 4, *Random* grouping can't provide obvious benefits. Parallel migration based on *R-Min-Cut*, on the other hand, reduces the communication volume by 24.7% even when the group size is 2. When increasing the group size to 32, the communication volume can be reduced by up to 41.2%.

2.3.3 Grouping and Shuffling Mechanisms

To estimate the benefits of our grouping and shuffling mechanisms, we use the same trace and metric as the above tests. We introduce *random* grouping as a control group to show the benefits of our optimized methods. The test results are shown in Figure 10. Note that in this test, we don't employ balanced grouping sizes. This

limitation is caused by the k-means grouping mechanism, since k-means can't ensure the clustering result sets have the same size. If the size of subgroups obtained from the grouping mechanisms are different, the comparison is not fair. To combat this, we use the group sizes from k-means to direct the execution of *R-Min-Cut* and *Random* mechanisms to ensure all the grouping mechanisms get the same number of subgroups. This also ensure that the size of the subgroups are identical.

From the tests, we have two key observations. First, both *R-Min-Cut* and *Kmeans-SF* outperform random grouping mechanism considerably. Specifically, *R-Min-Cut* outperforms random grouping by 37.0% on average and up to 74.4% when the total subgroup number is 4. *Kmeans-SF* also outperforms random grouping by 32.1% on average and up to 66.9%. Second, the k-means algorithm provides no obvious benefits without optimizing the order in which each subgroup is migrated. The tests show that for k-means, the shuffling mechanism reduces the communication volume by 35.3% on average, and up to 80%. This group of trace-based analysis proves that grouping mechanisms combined with a shuffling mechanism can considerably reduce the wide area communication volume during migration.

Additionally, in Figure 10 we can see that when k equals 8, the communication volume is even larger than when k equals 16. The result seems to contradict with the tests in Figure 9. The reason is that k-means clustering causes varying size of subgroups with different values of k.

2.3.4 Simulation of MPI Performance over WAN

To verify the practicability of *Clique Migration* in production environments, we test our optimization mechanisms on a simulated inter-cloud platform. The configurations of the system have been described at the beginning of this section. In this group of tests, we aim to first quantify the benefits of *R-Min-Cut* and *Kmeans-SF* al-

gorithms in a real scenario. Then, explore the range of application of each algorithm. Finally, we provide some valuable observations made from the tests.

To quantitatively analyze application performance during migration via a net-

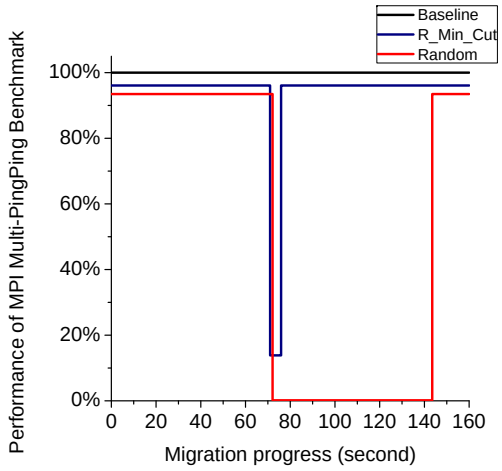


Fig. 11. The performance of MPI1 parallel transfer Multi-PingPing benchmark during migration normalized to Non-Migration.

work with limited bandwidth, we run MPI1 parallel transfer benchmark *Multi-PingPing* on 8 VMs. We first obtain the traffic statistics by tracing the benchmarks, and forming a traffic matrix. Then, the traffic matrix is used as input to *R-Min-Cut* and *Kmeans-SF* algorithms. The algorithms will make optimization decisions, resulting in the VMs contained in each subgroups and the order in which each subgroup should be migrated. *Multi-PingPing* contains multiple pairs of VMs collaborating on the tasks, with each pair of VMs forming a natural subgroup. We perform *R-Min-Cut* and *Kmeans-SF* algorithms on the traffic matrix of 8 VMs using group size of 4, they both generate exactly the same migration decision. The first migration subgroup is {VM0, VM1, VM2, VM3}, followed by {VM4, VM5, VM6, VM7}. The *Multi-PingPing* application contains 4 process groups, {VM0, VM1}, {VM2, VM3}, {VM4, VM5}, and {VM6, VM7}. VMs contained in the same process group have

intensive communication, while inter-group VMs have little communication. As a result, the decision made by our optimized grouping mechanisms can avoid the intensive intra-group communication during migration. The test results are shown in Figure 11. As the four process groups demonstrate similar performance patterns, for the regularity of the figure, we just show the throughput of group1 which contains VM0 and VM1 in the figure. VM0 and VM1 have migration time of 71.4s and 72.8s in the migration under the direction of *R-Min-Cut* algorithm. While in the scenario of *Random* grouping, VM0 and VM1 happen to be allocated to different migration subgroups. VM0 and VM1 have migration time of 72.1s and 71.5s respectively. The tests show that in the *Random* grouping, during the first 72.1s, VM0 and VM1 located at the same data center, and connected through LAN, the throughput between them is 43.0 MB/s. After 72.1s, VM0 has been migrated to the other simulated data center, and VM0 and VM1 are connected through the link with bandwidth configured as 100 Mbps. As a result, after 72.1s, the application performance undergo serious degradation, and the throughput decreases to 0.07 MB/s. After 143.6s, both VM0 and VM1 are migrated to the destination data center, and the throughput of the application increased to the normal level of 43 MB/s. For the process group which contains VM0 and VM1, migration based on *Random* grouping causes a performance degradation period of 71.4 seconds. In comparison, migration based on *R-Min-Cut* only causes a performance degradation period of 4.9 seconds. Theoretically, *R-Min-Cut* should not cause performance degradation during migration. However, although VM0 and VM1 are migrated in parallel, the completion time of these two VMs has a gap of 1.4s, which causes performance degradation.

In the above scenario, *Random* grouping will split some tightly related VMs into different migration subgroups. *R-Min-Cut*, on the other hand, can ensure all the VM pairs with strong traffic affinity are allocated to the same migration subgroup.

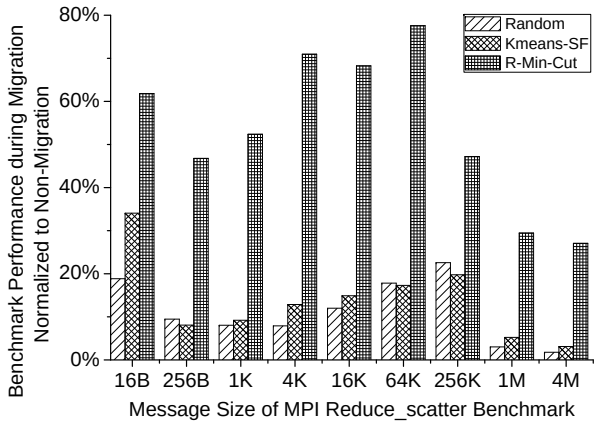


Fig. 12. The Performance of MPI1 collective *Reduce_scatter* benchmark with different message sizes during migration.

To further verify the usefulness of *R-Min-Cut* and *Kmeans-SF* algorithms in complex environments, we run Intel MPI1 collective benchmark *Reduce_scatter* on 32 VMs hosted by 8 physical machines. We first run the benchmark in static environments and get the traffic matrix of the VMs. Then we perform *R-Min-Cut* and *Kmeans-SF* algorithms on the matrix respectively to get the migration decisions. Finally, we perform migration in our simulated inter-cloud platform. In this test, we set the group number as 8. The test results are shown in Figure 12.

To normalize the application performance during migration, we use the reciprocal of response time as the metric. The reciprocal of response time in the non-migration scenario is used as baseline. For example, when the message size is 1KB, in the non-migration scenario, the average response time of a single Reduce_scatter operation costs 0.8 ms. When the migration is directed by *random grouping*, *Kmeans-SF* and *R-Min-Cut*, the response time is 9.9 ms, 8.7 ms and 1.5 ms respectively. Thus the normalized performance of the three mechanisms is 8.0%, 9.2%, and 52.4% of the baseline performance respectively.

The tests show that *R-Min-Cut* can keep the application performance between 27.1% and 77.8% of the baseline performance. *K-menas-SF* doesn't performs as well as it does in the *Multi-PingPing* scenario. We also observed an interesting phenomenon in our tests. During migration, when the message size if less than 1KB, the maximum response time can be up to 7 times of the average response time. This implies that in the environments with limited network bandwidth, the transferring of small message may encounter large transfer fluctuations. Therefore, in a scenario where high QoS is required, some mechanisms are needed to reduce the ratio of this kind of fluctuation.

CHAPTER 3

PROACTIVE CACHE WARM-UP OF DESTINATION HOSTS IN VM MIGRATION CONTEXTS

In virtualization platforms, host-side storage caches can serve virtual machines (VM) disk I/O requests, which originally target network storage servers. When these requests hit host-side caches, network and disk access latencies are obviated, and thus VMs perceive improved storage performance. VM migration is common in cloud environments, however, VM migration does not transfer host-side cache states. As a result, a newly migrated VM suffers performance degradation until the cache is fully rebuilt. The performance degradation period can be hours long if the cache is *naturally* warmed up. Employing existing cache warm-up solutions such as *migrating host-side cache* and *Bonfire*, VMs may either have a prolonged total migration time or undergo a performance degradation period of tens of minutes due to the warm-up caused storage contention. We propose *Successor*, which proactively warms up caches of destination hosts before migration completes. Specifically, accessibility of destination hosts during migration enables *Successor* to parallelize cache warm-up and VM migration. Compared with *migrating host-side cache* and *Bonfire*, *Successor* achieves zero VM-perceived cache warm-up time with low resource costs and performance penalties. We have implemented a prototype of *Successor* on QEMU/KVM based virtualization platform and verified its efficiency.

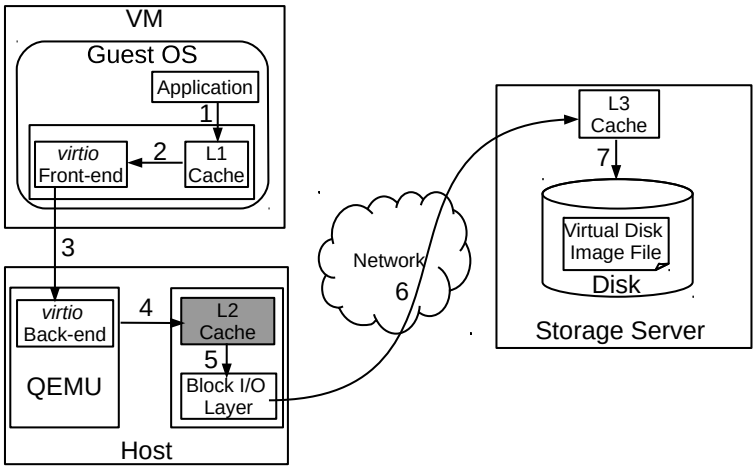


Fig. 13. A simplified sequence of requests to fulfill I/O operations issued by VM applications. The arrows represent logic sequences.

3.1 Problem Statement

Our work parallelizes cache warm-up of the destination host and VM migration so that a newly migrated VM perceives a warm cache instantly after migration completes. In terms of VM storage performance, we define the optimal host-side cache warm-up in VM migration contexts as follows.

Definition. Optimal Host-side Cache Warm-up Mechanism in VM Migration Contexts. Let D be the VM storage I/O latency distributions, and L the VM migration latency. Then a host-side cache warm-up mechanism W is optimal if the migrated VM instantly perceives unchanged D when migration completes, and L is not prolonged by W.

We first discuss the importance of warm host-side cache. Then, we discuss the importance of eliminating VM-perceived cache warm-up period as well as limitations of existing cache warm-up solutions. Finally, we discuss the optimal solution.

Table 3. CloudVPS traces. WSS denotes the VM working set size, which is the total data size of unique read or write. Reaccess represents the fraction of repeated read, including read after read and read after write.

VM	IOPS		Volume(GB)		WSS(GB)	Reaccess
	Read	Write	Read	Write		
vps26020	1	1.1	6.65	1.57	0.63	0.96
vps26107	17.7	18.4	36.98	33.42	9.45	0.81
vps26134	0.9	4.4	2.41	5.57	1.83	0.46
vps26136	1.6	11.7	3.03	33.61	2.83	0.68
vps26148	8.2	7.5	6.46	11.51	3.48	0.78
vps26215	3.8	17.1	15.20	24.28	12.12	0.41
vps26254	1.3	0.9	3.28	4.17	2.92	0.30
vps26255	1.6	11.8	6.05	18.44	6.23	0.12
vps26271	0.2	0.6	1.29	1.36	1.00	0.48
vps26330	1.6	13.6	2.77	31.92	1.45	0.79
vps26356	0.4	0.9	1.77	0.88	1.13	0.49
vps26391	0.6	5.7	1.31	7.94	1.56	0.37
vps26401	2.1	2.4	9.06	5.65	6.77	0.49
vps26440	3.1	2.2	10.98	3.56	1.48	0.90
vps26458	0.6	3.1	2.69	3.84	1.90	0.46
vps26477	0.2	0.15	0.53	0.37	0.48	0.53
vps26511	12.1	15.2	16.42	19.37	6.38	0.65
vps26535	8.0	18.5	36.53	35.79	20.80	0.61
Average	3.6	7.5	9.08	13.51	4.58	0.80

3.1.1 Importance of Warm Host-side (L2) Caches

Figure 13[1] shows the critical path of storage access in virtualization systems. From the viewpoint of applications running in VMs, the VM direct cache (L1), host-side cache (L2), and storage-side cache (L3) compose a multi-level cache.

We analyze two-day traces [4] of *CloudVPS* production VMs. The traces are collected at block I/O layers under L2 caches. As it's shown in Table 3, VMs such as *vps26440* has a read volume which is 7.4x of its WSS; *vps26107*, *vps26215* and *vps26535* have large read volumes of tens of GB. Due to the lack of simultaneously collected traces at L1 and L2, it's not feasible to make a quantitative comparison between the hit ratio of L1 and L2 caches. Also, the miss ratio of L1 caches cannot be quantitatively demonstrated. However, considering the *CloudVPS* traces are filtered by L1 and L2 caches, these large read volumes probed under L2 caches are clear proof that considerable I/O requests miss at L2 caches and more I/O requests miss at L1 caches. Thus, reducing the miss penalty of L1 caches is critical for improving read I/O performance.

Increasing hit rates of L2 caches is the most straightforward and efficient way to reduce the miss penalty of L1 caches. As it's shown in Table 4, once L2 cache hits, VMs perceive a nearly 8x read throughput increment. Once L2 cache misses, L3 cache helps improving application performance because disk accesses can be obviated. However, L3 caches only provide obvious benefits in the case that disks of the storage server are under pressure. If the storage server is idle, L3 caches only increase VMs read throughput by 20%, because the block I/O layer overhead and the network latency involved in L3 cache access are comparable to disk access time.

[1]L1, L2, and L3 are commonly used to denote CPU data cache hierarchies. In the virtualization storage stack, we use L1, L2, and L3 to denote the VM, host, and storage server side cache, respectively.

Table 4. Impacts of the hypervisor layer cache on the VM Storage I/O performance. *fio randread* benchmark is executed upon a 10 GB file in a VM under different host-side and storage server-side cache states.

Storage Server	Hypervisor	VM IOPS
L3 cache miss & disk busy	L2 cache miss	123
L3 cache miss & disk idle	L2 cache miss	495
L3 cache hit	L2 cache miss	597
Not Accessed	L2 cache hit	3883

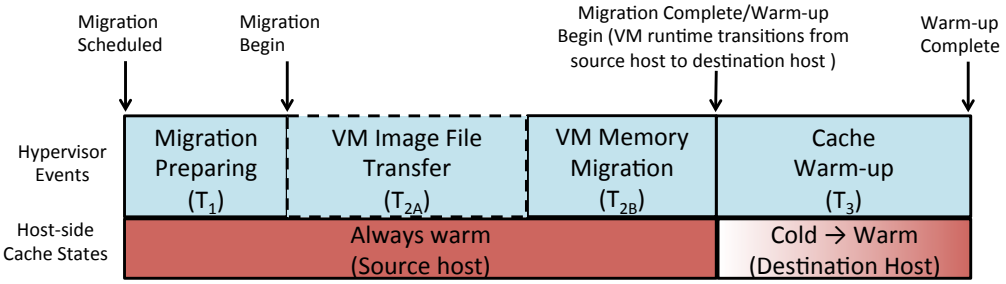


Fig. 14. *Migration-Then-Warmup.* T_{2A} exists only in storage migration scenarios, since memory-only migration doesn't transfer VM image files.

In production systems, the small capacity of L1 caches and high access overhead of L3 caches are obvious limitations for improving VM storage performance. L2 caches become standard components for optimizing VM storage performance. Cache pooling [31] and SSD based caching [15, 52] have been deployed to extend the capacities of L2 caches so as to promote hit rates. In general, the warmness of L2 caches provides the following benefits. First, it improves the VM-perceived storage performance. Second, it reduces network traffic and disk load of storage servers. Third, it mitigates the impact of storage contention on VM performance. Finally, it accelerates the VM startup.

3.1.2 Importance of Eliminating VM-Perceived Cache Warm-up Period

As it's shown in Figure 14, intuitively, the cache of the destination host gradually warms up after migration completes and the VM resumes running on the des-

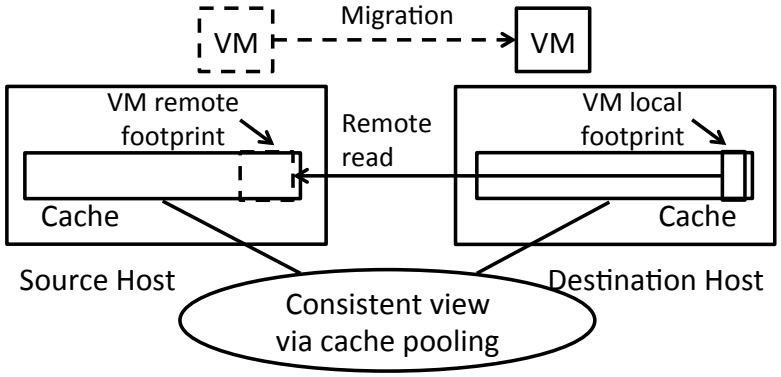


Fig. 15. VM migration with cache pooling.

tination machine. The problem of *migration-then-warmup* is that the migrated VM will perceive the T_3 period shown in Figure 14, and during T_3, the VM will undergo low storage performance due to HDD access and storage contention caused by cache warm-up. Although during T_3 the VM will observe low storage performance, if T_3 is short, this is not a serious problem. However, our analysis on *Cloud VPS* traces shows that the working set sizes of I/O intensive VMs can be up to 20 GB, which can cause a T_3 of more than 15 minutes. Thus, we argue that it's imperative to eliminate T_3.

3.1.3 Existing Warm-up Solutions and Their Limitations

3.1.3.1 Cache Pooling

Cache pooling is a coldness-avoidance solution. Cache pooling enables the destination host to access the source host's pages after VM migration, so as to obviate the need to migrate the cached pages from the source host to the destination host and achieve seamless VM migration [10]. Figure 15 demonstrates how cache pooling handles VM migration between two hosts. When the migrated VM issues a read request that causes a cache miss on the destination host's cache, the destination host

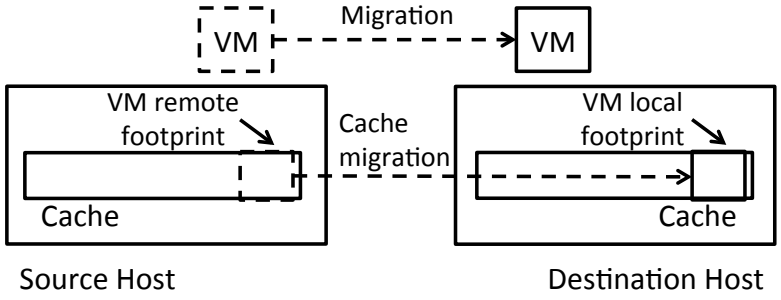


Fig. 16. VM migration with host-side cache transfer.

forwards the read to the cache pool, which fetches the page from the source host via remote paging instead of accessing the storage server.

Cache pool is not a standard component of virtualization platforms. Additionally, cache pooling has two main limitations. First, the performance of remote paging is much lower than accessing the local cache, because remote reads incur additional network latencies. The post-migration performance degradation of VMs with cache pooling is reported in [10]. Second, considering network latencies, cache pooling is usually implemented in local area environments. Thus, for wide area migration, cache pooling is not a practical solution.

3.1.3.2 Migrating the Host-side Cache

A straightforward method to maintain warmness of host-side caches is making cache transfer a procedure of VM migration process. As it's shown in Figure 16, during VM migration, the hypervisor can transfer the host-side cache before the VM resumes running on the destination host. Transferring the host-side cache maintains the cache warmness after VM migration. However, host-side cache migration is not a default function of most mainstream hypervisors. Currently, only vMotion of a vFRC-enabled VM supports transferring host-side caches [70]. KVM and Xen don't support transferring host-side caches. Additionally, as we have demonstrated using

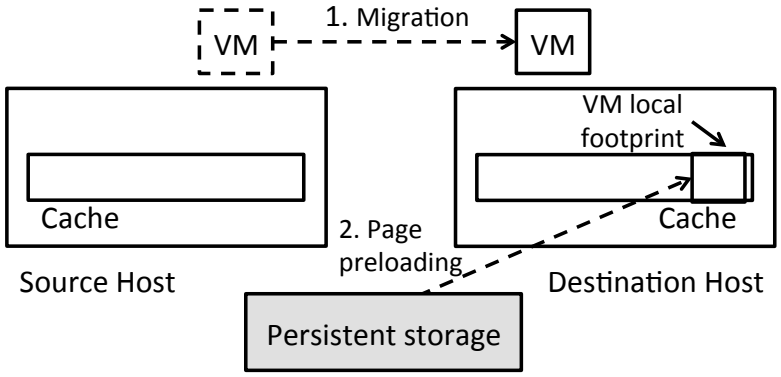


Fig. 17. VM migration with Bonfire cache warm-up.

CloudVPS traces, VM working set size can be tens of GB, moving such a large bulk of data during migration will considerably prolong the total VM migration latency, which is a key metric to measure migration performance, thus, makes this mechanism suboptimal.

3.1.3.3 Bonfire

Bonfire-like tools [75] can be used in virtualization environments to warm up host-side caches. As it's shown in Figure 17, after VM migration completes, *Bonfire* prefetches pages from persistent storage to warm up the cache of the destination host. As a host-side cache warm-up tool, *Bonfire* has two limitations. First, although *Bonfire* accelerates the cache warm-up process, VMs still observe a relatively long cache warm-up period when the cache is large. Moreover, our tests show that the VM IOPS decreases to less than 20 during the warm-up period, which is a serious performance problem. Thus, *Bonfire* violates the optimal host-side cache warm-up requirement that a VM should perceive unchanged storage performance instantly after migration completes. Second, *Bonfire* prefetches data from storage devices into caches. This warm-up I/O path is suboptimal in storage migration scenario. Since

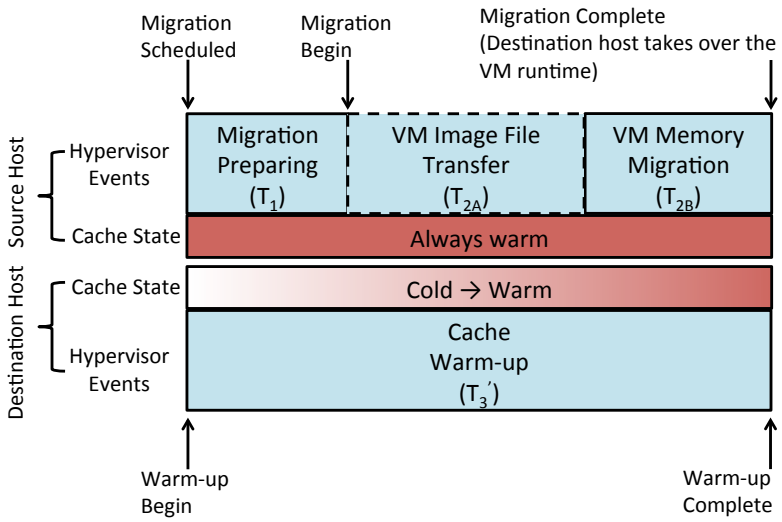


Fig. 18. Migration-And-Warmup.

the whole virtual disk will be transferred from the source host to the destination host, the candidate pages can be admitted into the cache during the transfer of virtual disks to avoid the succedent data prefetching from the disks.

3.1.4 Optimal Solution: Proactive Cache Warm-up

The reason that existing solutions fail to achieve optimal host-side cache warm-up is that the warm-up process and the migration process are not parallelized. To achieve optimal warm-up, instead of waiting until the VM migration completes, cache warm-up can be conducted proactively as soon as migration is scheduled. As it's shown in Figure 18, *Migration-And-Warmup* scheme eliminates T3. Specifically, T3 in Figure 18 can be overlapped with T1 and T2, and become T3'.

The challenges for proactive cache warm-up are twofold. First, there is a *write-after-prefetch* cache consistency problem in memory-only migration scenario. Since a VM is still running on the source host during migration, a page prefetched into the cache of the destination host can be updated by a source side write request, which

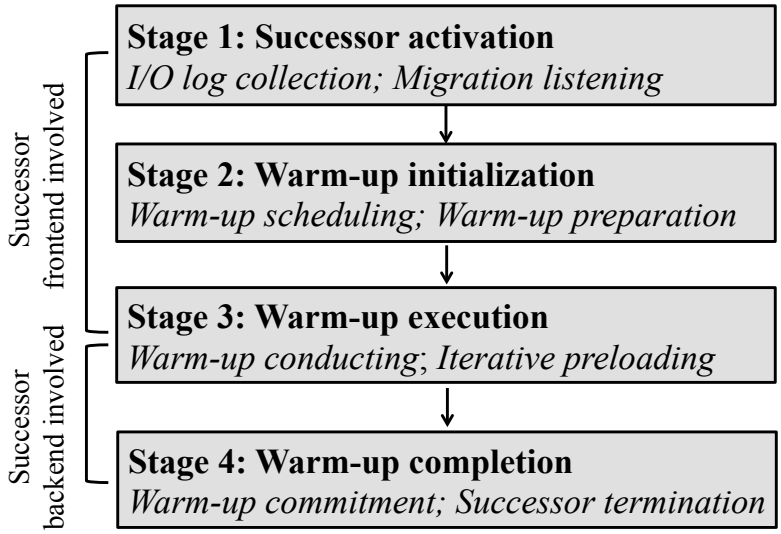


Fig. 19. Successor cache warm-up timeline.

makes the prefetched page in the destination cache become stale. Second, there is a warm-up I/O path problem for proactive cache warm-up in storage migration scenario. During storage migration, the virtual disk still resides at the source side, thus, the destination host is not able to build local cache footprint of the migrating VM before migration completes.

We propose *Successor*, a proactive cache warm-up solution. *Successor* employs *dirty page tracking* and *piggyback warm-up on migration* to solve write-after-prefetch cache consistency problem and absence of data source problem, respectively.

3.2 Successor

3.2.1 Design Overview

Successor is designed as a plug-in component of virtualization platforms. The logical steps that we warm up a destination cache using *Successor* are summarized in Figure 19.

Stage 1: Successor activation We begin with activating *Successor* in the

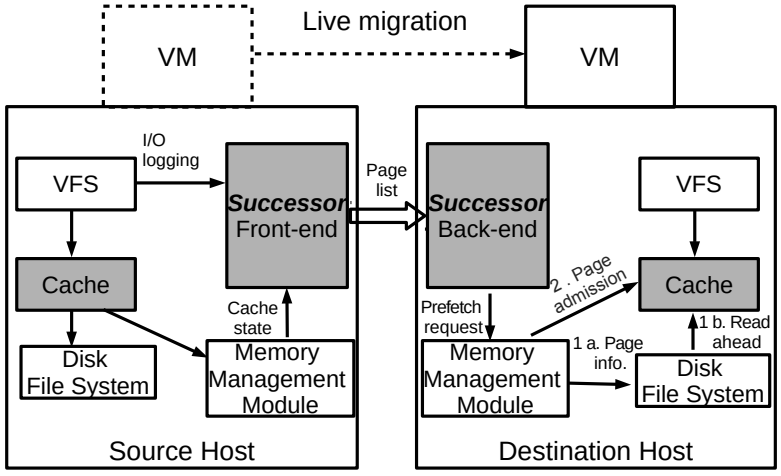


Fig. 20. Successor architecture.

source and destination hosts. Two subroutines, I/O logging and migration listening, of the *Successor fronend* are initiated in the source host. In this stage, *Successor backend* in the destination host lazily waits for the warm-up execution request from the frontend.

Stage 2: Warm-up initialization Upon the scheduling of a VM migration, *Successor frontend* determines the optimal time to initiate warm-up. *Successor frontend* forms an optimal candidate page list and sends it to *Successor backend*.

Stage 3: Warm-up execution Upon receiving a page list, *Successor backend* preloads the pages into the destination cache. During each round of prefetch, *Successor frontend* employs dirty page tracking to record the pages which are updated after prefetch. Page preloading is an iterative process. The *write-after-prefetch* pages in each round will be dropped from the destination cache and fetched again in the next round.

Stage 4: Warm-up completion Once VM migration completes, *Successor backend* conducts the last round page preloading to ensure the consistency of the destination cache. *Successor backend* notifies *Successor frontend* that warm-up work

is done, so the frontend can terminate I/O logging and migration listening of the migrated VM.

3.2.2 Successor Architecture

Figure 20 presents the *Successor* architecture and its interaction with other system components. *Successor* consists of a frontend component and a backend component. *Successor frontend* functions as a warm-up coordinator, which conducts I/O log collection, migration listening, and cache page checking to form a candidate page list for warm-up. The frontend also takes the VM migration scheduling information, such as migration preparing time, image file transfer time, and memory migration time, into consideration to determine the optimal time to initiate warm-up. The *Successor backend* functions as a warm-up executor, which preloads candidate pages into the destination cache.

3.2.2.1 Successor frontend

Successor frontend includes the following routines.

Source cache snapshotting. The most straightforward candidate pages for warm-up are the existing contents of the source host caches. Instead of copying those contents in bulk from the source host to the destination host, *Successor frontend* communicates with the memory management module via system calls like *mincore* to gain a snapshot which contains a list of page numbers indicating cache contents. With this page list, pages can be recognized and fetched from the storage server into the destination cache during warm-up process.

I/O logging. Getting a list of the pages residing in the source cache is not enough to make an optimal warm-up decision, since it's not guaranteed that the size of the destination cache is exactly the same as the source cache. Page access history

is needed to decide which pages should be pruned if the destination cache is smaller than the source cache, or which additional pages should be preloaded if there is more cache space available at the destination side. To avoid the I/O requests being filtered by the cache, page access requests need to be probed above the cache. Virtual file system (VFS) layer is a suitable log collection point since it's above the cache and VFS requests contain *access type, offset, size* information. The I/O logging routine probes the accesses on specified virtual disk files and extracts the access information of each request into a log file.

I/O log based cache alignment. Using the snapshot of the source cache and the I/O log, an optimal candidate page list can be formed. Since both the source and the destination host use LRU as the cache replacement algorithm, *Successor* uses recency as the criteria to decide the hotness of pages. Pruning the source cache snapshot or adding additional pages, *Successor* prefers to reserve the most recently accessed pages.

Dirty page tracking. *Successor* employs dirty page tracking based iterative preloading to avoid the write-after-prefetch cache consistency problem. Since all of the write requests are probed by the I/O logging routine of the frontend, dirty pages can be determined through comparing the write time of a page in the source host and the preloading time of the page in the destination host. During each round of preloading, a dirty page list is formed and pushed to the backend for the next round of preloading. The last round of preloading is performed instantly after VM migration completes.

3.2.2.2 Successor backend

Successor backend mainly includes page preloading and piggyback cache warm-up routines for memory-only migration and storage migration, respectively.

Page preloading is the cache warm-up routine in memory-only migration scenario. Once a page list is received from the frontend, the backend preloads the pages from the storage server into the cache of the destination host. The page preloading path is indicated by 1b in Figure 20. Since reading the pages incurs pressure on the storage server, *Successor* supports preloading at various rates.

Piggyback warm-up on migration is the cache warm-up routine for storage migration scenario. For storage migration, virtual disk image files are transferred from source hosts to destination hosts. During migration, all of the virtual disk data including the hot pages pass through the cache of the destination host. *Successor* backend selectively admits hot pages into caches on-the-fly, thus, no further data preloading from the storage server is needed. The path 2 in Figure 20 shows the backend processing flow during storage migration.

Both *page preloading* in the memory-only migration, and *piggyback warm-up on migration* in the storage migration scenario can be implemented via system call like *fadvise*.

3.3 Implementation and Evaluation

Our tests are conducted on a 19-nodes cluster interconnected with a 24-port HP 1810-24G switch. Each physical node has two 2.30 GHz Intel(R) Xeon(R) CPU E5-2630 processors, totally 64 GB DRAM, one 500 GB hard drive. One 12 TB RAID6 volume, which is managed by an LSI 9271-8i RAID controller, serves as the NAS storage of the cluster. The host OS is CentOS with Linux kernel version 2.6.32, with QEMU and KVM deployed as the hypervisor. We employ two types of VMs to conduct the tests, one with a 10 GB virtual disk to run micro-benchmarks, and the other with an 80 GB virtual disk to conduct macro-benchmarks. VMs are deployed with 2 GB memory and 2 vCPU. We implemented *Successor* prototype as a user space tool

Table 5. RunTime of YCSB workloads under different host-side cache states. Each workload conducts a total of 100K operations on 100K records (Unit: Second).

Workload	Host-side Cache/ Storage Server Cache	
	Warm/Not Accessed	Cold/Warm
Workload A	177	600
Workload B	258	1280
Workload C	176	1264
Workload D	234	1000
Workload E	991	2200

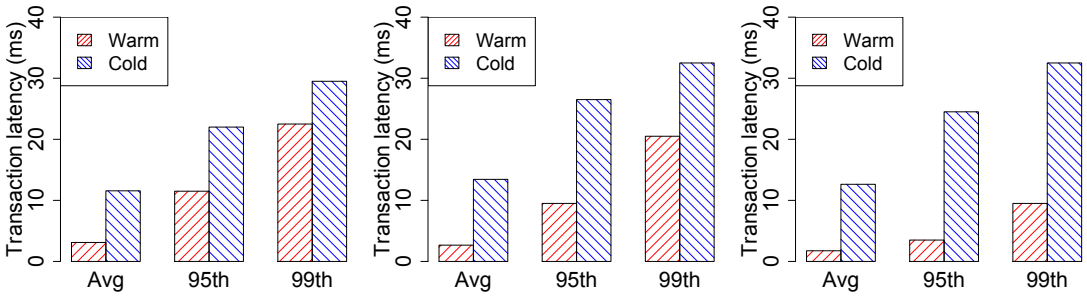
at the host layer. We employ DRAM as the host-side cache.

Our evaluation employs a mix of YCSB [21] macro-benchmarks, *fio* [5] micro-benchmarks, and *CloudVPS* production storage traces [4]. First, we evaluate the benefits of warm host-side caches using five representative cloud workloads of YCSB macro-benchmark. Second, to focus on evaluating the performance of VM storage subsystem in migration contexts and accurately compare the performance of *Successor* with other schemes, we employ *fio* micro-benchmark to minimize interferences caused by other system resources including CPU and memory. Third, we compare the post-migration VM storage performance under different cache warm-up mechanisms through replaying IaaS VM traces, which represent the storage characteristics of production platforms. Finally, we evaluate the overheads of *Successor*.

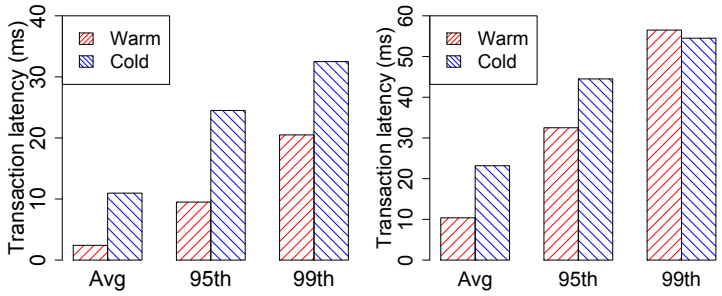
3.3.1 The benefits of warm host-side cache

3.3.1.1 Improving VM read performance

We first compare the VM read performance under warm and cold host-side caches using YCSB benchmark. In all tests, we use gigabit Ethernet. Each workload conducts a total of 100K operations on 100K records. Other parameters such as the read and write ratios are configured as the default values. The runtimes of workloads under



(a) Workload A READ (b) Workload B READ (c) Workload C READ



(d) Workload D READ (e) Workload E SCAN

Fig. 21. Performance of YCSB workloads under warm and cold host-side caches, respectively. Each workload may have a mix of reads and writes. Since the warmness of host-side caches has ignorable impact on write requests such as *update* and *insert*, we focus on the performance of *READ* operation of Workload A, B, C, D, and *SCAN* operation of Workload E.

warm and cold host-side caches are shown in Table 5. Figure 21 shows the latency distributions in the scenarios that the host-side cache is warm and cold, respectively. To make the host-side cache cold and warm, we drop the cached pages and run the workload once before tests, respectively. In cold host-side cache tests, the storage-side cache is actually warm since we don't drop the cache of storage server explicitly. In general, under warm host-side cache, a VM perceives a lower average read latency, as well as 95th and 99th percentile latencies. Specifically, compared with the cold cache scenario, warm caches reduce average read latency by 55% to 86%. Compared with average latency, the improvement of tail latencies is moderate, but for *Workload*

C, a read-only workload, the 95th and 99th percentile latencies still reduce by 86% and 71%, respectively. This group of tests shows that warm host-side caches improve the VM read performance a lot. Thus, from the viewpoint of applications running in VMs, it's important to maintain the warmness of the caches.

3.3.1.2 Reducing traffic and load on storage servers

Warm host-side caches can absorb I/O requests which originally target the network storage servers, reducing network traffic and storage server load. Figure 22 shows the network traffic rate between the host machine and the storage server when the host-side cache is warm and cold, respectively. In the tests, we keep the cache of storage server warm to avoid the network utilization being limited by disk speed. Compared with the cold cache, warm host-side cache reduces the network traffic rate between the host and the storage server by 81% for *Workload E* and up to 95% for *Workload C*. When network traffic arrives on storage servers, if the requested data is not included in the storage-side cache, disk access is required. Our tests show that compared with warm host-side caches, cold ones cause about a 10% increment of storage server disk utilization under the pressure of a single workload execution.

3.3.1.3 Storage contention resisting and VM startup accelerating

Disk I/O is an important factor which limits the growth of VM density. When multiple VMs share storage, VM storage performance is unpredictable during I/O contention. Warm host-side cache helps to resist storage contention. Figure 23 shows the VM storage performance under different pressure levels when the host-side cache is cold and warm, respectively. We have two observations. First, warm caches make VM storage performance 23x better than cold caches. Second, cold caches increase the VM average I/O latency by up to 81% as the storage pressure increases; warm

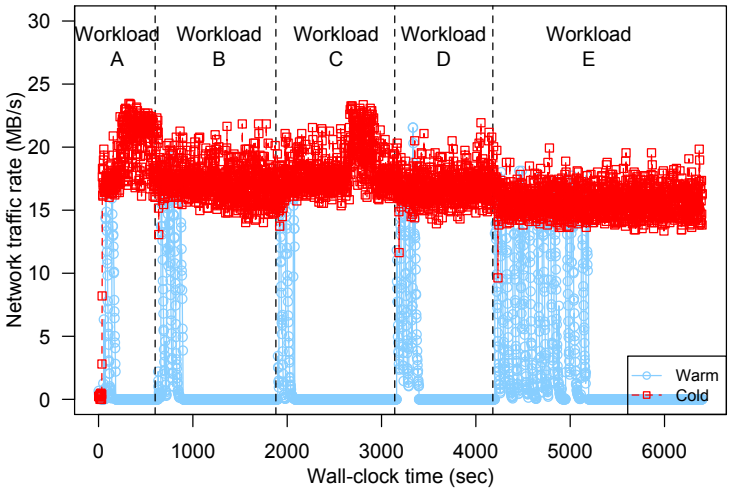


Fig. 22. Network traffic rate between the host and storage server under warm and cold host-side caches, respectively.

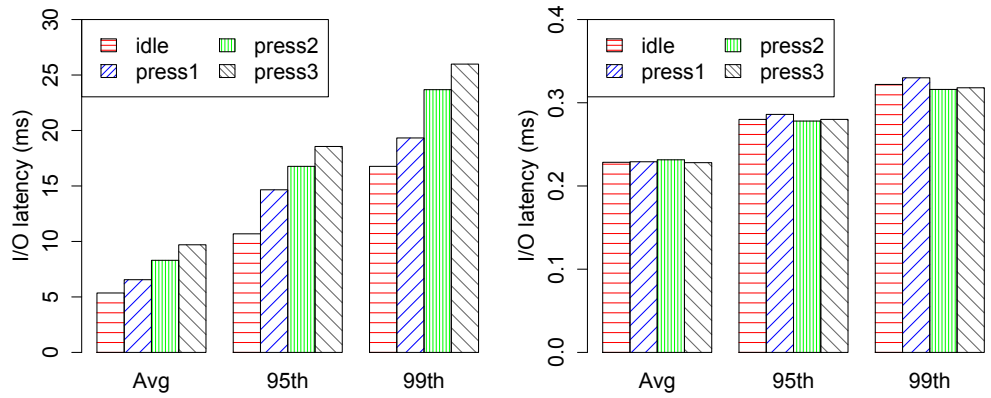
caches make VMs perceive consistent high performance, which is not affected by I/O activities of other VMs.

Our tests also show that a warm host-side cache accelerates the startup of a single VM by about 20%, although the VM startup is CPU intensive. In the offline migration scenario, VMs are migrated then restarted. As a result, proactively fetching the boot demanded data into the cache can reduce the VM startup time. Especially when multiple VMs are migrated in batch and restarted simultaneously as in [77, 49].

3.3.2 VM storage performance in migration contexts

3.3.2.1 Post-migration storage performance degradation

We first evaluate application performance degradation after VM migration using three synthetic random read workloads: *Uniform*, *Zipf*, and *Pareto*. Each workload has 2.5 million 4KB candidate pages, with a working set size of 10, 1.8, and 5.6 GB for *Uniform*, *Zipf*, and *Pareto*, respectively. We use IOPS, and 95th percentile



(a) Cold host-side cache. (b) Warm host-side cache.

Fig. 23. The performance of *fio randread* workload under different host-side cache states and storage server pressure levels. Two VMs are running concurrently on two physical machines which share a storage server. The measured VM executes a single *fio* job, the other VM presses the storage system using 1, 2, or 4 *fio* jobs to impose different levels of storage pressure.

latency to measure the throughput and response time of I/O requests. Benchmarks are running in VMs. Since we focus on evaluating the impact of host-side cache on the VM disk storage, direct I/O is set for the benchmarks to bypass the VM direct memory. In each run, 2.5 million read requests are issued to warm up the host-side cache. A second round benchmarking measures the storage performance of VM under warm host-side cache. Then, VM migration is performed and another round of benchmarking is conducted to measure the post-migration VM storage performance. Compared with pre-migration VM storage performance under warm host-side cache, the post-migration throughputs of benchmarks drop by 65.3% to 96.2%. Meanwhile, the 95th percentile latencies increase by 10x to 37x.

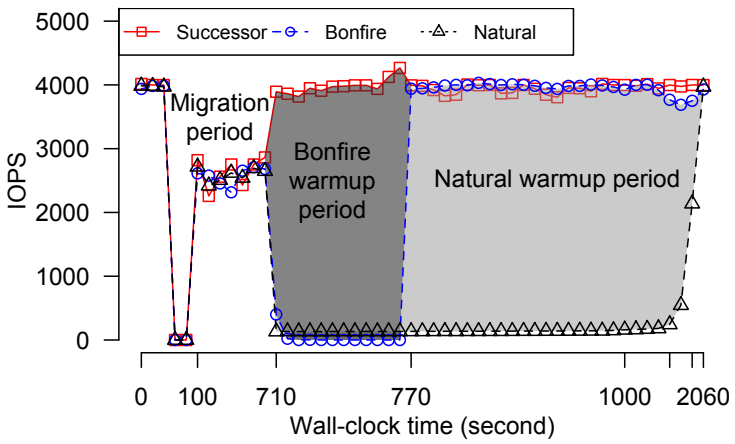


Fig. 24. The on-the-fly VM storage performance before, during, and after migration with *Successor*, *Bonfire* and *Natural* cache warm-up mechanisms.

3.3.2.2 VM-perceived cache warm-up time

We evaluate the efficacy of *Successor* through comparing it with Bonfire [75], a state-of-the-art cache warm-up solution. We focus on measuring the VM-perceived cache warm-up time. To focus on VM storage subsystem, and to give a clear picture of the VM storage performance fluctuation in migration contexts, we execute an *fio uniform randread* benchmark inside a VM. In this group of tests, we conduct storage migration. The VM is configured with a 10 GB virtual disk, and the working set of the *fio* benchmark is 2 GB. We run the benchmark once to warm up the cache of the source host, then we run the benchmark for a second time, meanwhile, we execute live storage migration of the VM. We monitor the benchmark performance before, during, and after migration. The cache of the destination host is warmed up using *Natural*, *Bonfire*, and *Successor* mechanisms respectively. *Natural* indicates the cache is warmed up on demand, no specific warm-up mechanism is employed.

From Figure 24, we have several important observations. First, if the cache is warmed up *naturally* without any specific mechanism, it takes more than 20 minutes

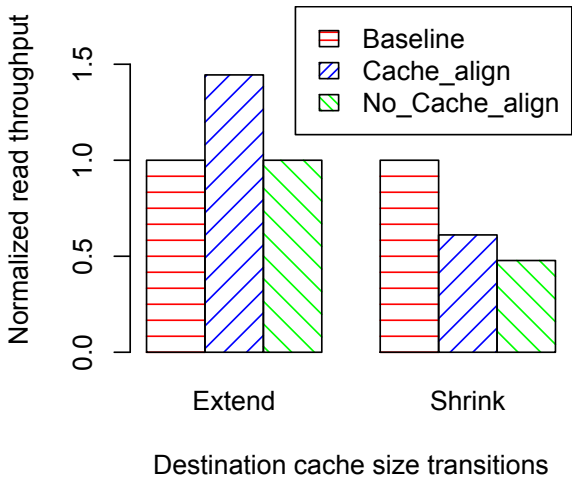
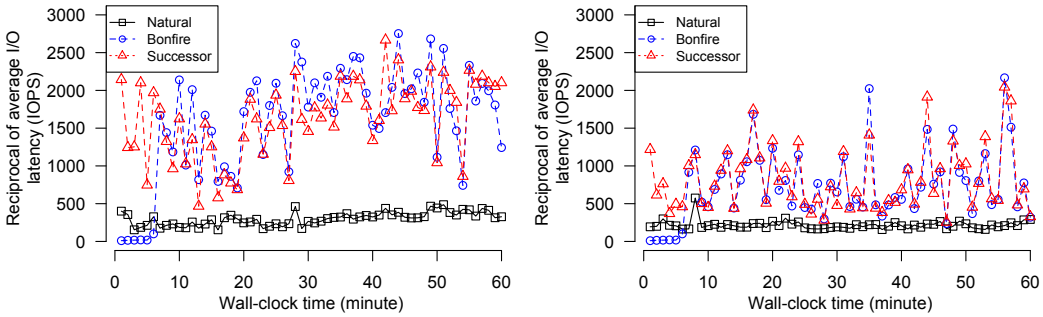


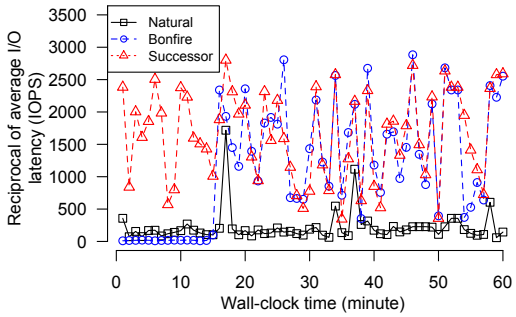
Fig. 25. The post-migration throughput of *Zipf* workload in asymmetric destination cache size scenario with and without I/O log-based optimization.

for the destination cache to achieve a similar hit rate as the pre-migration source cache. The *Natural* cache warm-up time relies on the request arrival rate. Intensive requests result in a shorter cache warm-up time. In our *Natural* case, we use the best achievable I/O rates of random read to warm up the cache. Second, *Bonfire* shortens the cache convergence time, but during the cache warm-up period, IOPS of *fio* running in the VM drops to less than 20. Some I/Os are even hung for up to 180 seconds. To explain this, we further check the host-side CPU utilization of processes when conducting cache warm-up with *Bonfire*. We notice that *Bonfire* causes the CPU utilization of *rpciod* process, an RPC multiplexer daemon in the critical path of remote packet transmission, increasing from 30% to 100%. Thus, VM I/Os are blocked. Third, *Successor* achieves zero VM-perceived cache warm-up time. Using *Successor*, the VM perceives consistent latency distributions before and after migration. Also, *Successor* doesn't prolong the VM migration latency. These observations verify that parallelizing the warm-up and migration process is practical and optimal host-side cache warm-up is achievable.



(a) vps26107.

(b) vps26215.



(c) vps26535.

Fig. 26. The first-hour post-migration read latency of Day 2 CloudVPS traces.

3.3.2.3 I/O log-based cache alignment

In unplanned memory-only migration scenario, it's possible that there is not sufficient time to preload all of the candidate pages into the destination cache. Besides, the destination host may allocate a smaller cache to a migrating VM than what it previously has in the source host. In these cases, some pages need to be pruned during warm-up. Also, the destination host may allocate a larger cache to a migrating VM, then additional pages which are currently not in the cache of the source host can also be preloaded into the destination cache. Determining which pages should be pruned or additionally preloaded is critical for post-migration VM storage performance. Figure 25 shows the post-migration throughput of *Zipf* workload in the asymmetric destination cache size scenario with and without I/O log-based optimiza-

tion. The size of source cache is 1 GB, we extend it to 1.5 GB, and shrink it to 0.5 GB at the destination side respectively to evaluate the benefits of I/O log-based cache alignment. The pre-migration throughput is used as the baseline. In *extend* scenario, with I/O log-based cache alignment optimization the VM throughput increases by 44%. In the *shrink* scenario, with and without optimization, the throughput of workload decreases by 39% and 52%, respectively. This group of tests shows that I/O log-based cache alignment can be used to preload pages in preferential ways.

3.3.3 Benefits of Successor for production clouds

To evaluate the potential benefits of *Successor* for production clouds, we compare the post-migration VM storage performance under different warm-up mechanisms via replaying IaaS VM traces. We select to replay the block I/O trace of VM *vps26107*, *vps26215*, and *vps26535*, which have the largest working set in the CloudVPS traces. Trace replay is conducted in the VM user space. Since the traces span two days, we use the *day-1* trace to warm up the cache, followed by conducting VM migration, then replay the *day-2* traces and measure the latency of read requests. Considering the fluctuation of VM performance during migration and the variety of I/O arrival rate during the trace wall-clock time, we don't replay traces during migration, instead, we start replaying the *day-2* traces after migration completes. We focus on observing the first hour post-migration read latency of VMs.

Figure 26 shows the trace replay results. First, the post-migration VM storage performance degrades by 82%, 73%, and 87% for *vps26107*, *vps26215*, and *vps26535*, respectively. This is attributed to post-migration cache coldness which causes cache hit rate loss of 91%, 77%, and 96% for *vps26107*, *vps26215*, and *vps26535*, correspondingly. Second, *Natural* cache warm-up improves VM storage performance very slowly. VM storage performance doesn't increase much at the end of the first hour.

Third, *Bonfire* can accelerate cache warm-up. It takes 327 seconds, 311 seconds, and 861 seconds to warm up *vps26107*, *vps26215*, and *vps26535*, respectively. However, during *Bonfire* warm-up periods, VMs undergo IOPS of less than 20, which is ultra-low. The I/O response time is highly fluctuating under *Bonfire*, during which we observed a single I/O was hung for 180 seconds. Fourth, *Successor* enables VMs instantly achieving the same post-migration storage performance as pre-migration, thus obviating the extreme VM storage performance penalty during the *Bonfire* warm-up. Compared with the *Natural* warm-up, *Successor* brings a 5.4x, 3.7x, and 7.5x VM performance improvement for *vps26107*, *vps26215*, and *vps26535*, respectively. This group of tests verifies the potential benefits of deploying *Successor* in production clouds where migration activities are frequent.

3.3.4 Successor Overheads

Overheads of *Successor* front-end mainly come from two sources: cache snapshotting and VFS I/O logging. To quantify the overheads, we first measure the cache snapshotting time with different cache sizes, and with different virtual disk sizes, respectively. Tests show that page snapshotting time is positively correlated with both the cache size and virtual disk size, but dominated by the virtual disk size. Although page snapshotting is resource consuming, the effect of it on system performance is not serious. Because the snapshotting period is short, only 5.3 seconds for an 80 GB virtual disk file with a 16 GB cache. 80 GB is an average storage size per VM in production environments [11]. Cache snapshot volume cost is about 2MB per GB cache footprint. VFS I/O logging is implemented using *SystemTap*, which incurs a very little system performance penalty. We choose *fio*, an I/O intensive application, and *Advanced Encryption Standard (AES)*, a computation intensive application to evaluate the impact of VFS I/O logging on application performance. Tests show that

VFS I/O logging incurs less than 2% performance penalty to *fio randread* benchmark, and incurs less than 1% performance penalty to *AES*. About storage space consumption of logging, 1 GB I/O consumes about 2 MB disk space.

Overheads of *Successor* back-end are scenario dependent. In storage migration scenario, the cache warm-up is implemented via *Piggyback warm-up on migration*, which is achieved by using *SystemTap* utility to probe the entrance of pages into the cache and the *fadvise* system call to admit or evict the page. The implementation causes a very slight performance penalty to co-locating applications. In memory-only migration scenario, hot pages need to be preloaded from the storage server into the cache. This process incurs considerable disk contention to co-locating applications. Proactively prefetching candidate pages into the cache at a moderate rate can mitigate this impact.

CHAPTER 4

SYSTEM-LEVEL OPTIMIZATION OF VIRTUAL I/O

As per-server CPU cores continuously increase, application density of virtualization platforms has been increased, imposing high pressure on storage systems. Layers of caches are deployed to improve storage performance. Owing to its manageability and transparency advantages, hypervisor-side caching is widely employed. However, hypervisor-side caches locate at the lower layer of VM disk filesystems, thus, I/O virtualization operations are still involved in the critical path of cache access. Virtual I/O is expensive in terms of round-trip time (RTT) between the front-end and the back-end as well as CPU cycles. Virtual I/O caps the throughput (IOPS) of hypervisor-side caches and incurs additional energy consumption. If requested data reside in the VM-side DRAM cache, I/O virtualization overheads are obviated. Therefore, VM applications observe fast I/O response and systems have reduced energy consumption. Fortunately, the bandwidth between the virtual I/O front-end and back-end is high, which provides a chance to prefetch some correlated data from the back-end (hypervisor-side) cache into the front-end (VM-side) cache when a back-end access is inevitable. We propose *Virtual I/O Prefetcher (VIP)*, which aims to bridge the performance and capacity gap between VM-side and hypervisor-side caches via virtual I/O front-end prefetching. Our trace based simulations demonstrate that compared with the sequential prefetching, VIP prefetching algorithm improves VM cache hit ratio by up to 70%.

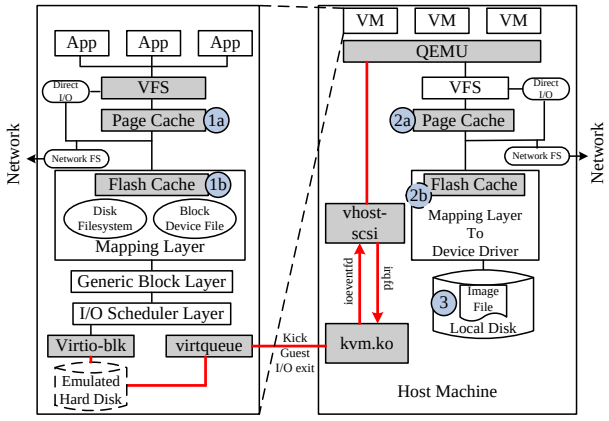


Fig. 27. System components involved in a VM block device operation. Page cache is a built-in module of most modern operating systems. Flash-based caches are optional but widely deployed in virtualization platforms to accelerate storage. The lines highlighted with red color represent virtual I/O path, which is CPU cycle consuming and imposes considerable latency on virtual I/O requests.

4.1 Motivation and Problem Statement

In this section, we first present storage access path of VMs, explain the involvement of storage components when cache hits at VM and hypervisor side, respectively. Second, we analyze where the virtual I/O overheads lie in. Third, we discuss existing optimizations on virtual I/O and their limitations. Finally, we discuss the benefits of virtual I/O front-end prefetching.

4.1.1 Storage Access of VMs

Block devices are commonly exposed to VMs via emulation. The host-side entity of a virtualized block device can be a file, an LVM logical volume, a device partition, or a whole device. Since device emulation incurs overheads, dedicated device allocation, also known as device passthrough, is implemented to enable a device being exclusively used by a VM without the involvement of I/O virtualization layers. How-

ever, not all devices can be allocated in the way of passthrough. For block devices, currently only the PCI-based devices such as PCIe SSDs can be assigned to VMs via *PCI passthrough*; SATA devices, however, cannot be allocated to VMs in the way of passthrough. In this paper we assume the persistent storage of VMs is backed by emulated block devices. But when we discuss the implementations of SSD-based guest-side caches, we will compare the performance and energy efficiency of SSDs connected to VMs via *PCI passthrough* and *virtio*, respectively.

For efficient emulation of block devices, paravirtualization is the standard solution. Two of the most widely used para-virtualized device drivers are *virtio* [60] and *Xen paravirtualization* [7]. The former is widely used in *QEMU/KVM* based virtualization platforms; the latter is from the *Xen* project. *Xen PV* and *virtio* are architecturally similar, we discuss the *virtio* based storage stack in details. As it's shown in Figure 27, assume a guest OS issued a *read* request on some disk file, the activities of the guest OS and the host OS components are as follows:

(1) The read request activates a Virtual Filesystem (VFS) function, passing to it a file descriptor and an offset.

(2) If the request doesn't indicate direct I/O, the VFS function determines whether the required data are available in the page cache *1a*. If *1a* hits, the data are returned from the cache and the request is completed.

(3) Assuming the page cache *1a* missed, the guest OS kernel must read the data from the block device. If the requested data reside in the flash cache, it's a flash cache hit at *1b*. In this case, data are fetched from the flash device and HDD access is obviated.

(4) Assuming it's a flash cache miss, the request has to go through the traditional generic block and I/O scheduler layer and then be served by the virtual I/O device

driver such as *virtio-blk*.

(5) Upon a read request, the *virtio-blk* frontend driver composes a request entry and places it into the descriptor table of the *virtqueue*. Then, the *virtio-blk* frontend driver will call *virtqueue_kick*, which causes guest I/O exit and triggers a hardware register access called *VIRTIO_PCI_QUEUE_NOTIFY*.

(6) *vhost* is the host-side *virtio* component for completing the virtual I/O request. Once *vhost* is notified by *KVM* for the guest kick, it fetches the *virtio* request from the queue and calls *QEMU*, which works as a regular userspace process, to complete the data transfer.

(7) To fetch the data, *QEMU* issues I/O requests which again traverse the host OS storage stack. Host-side *page cache* or the optional *flash cache* are successively checked. Once the data hit at the caches or have been fetched from the HDD, *vhost* updates the *status* bit of the *virtio* request and issues an *irqfd* interrupt to notify the guest that the request is completed.

4.1.2 I/O Virtualization Overheads

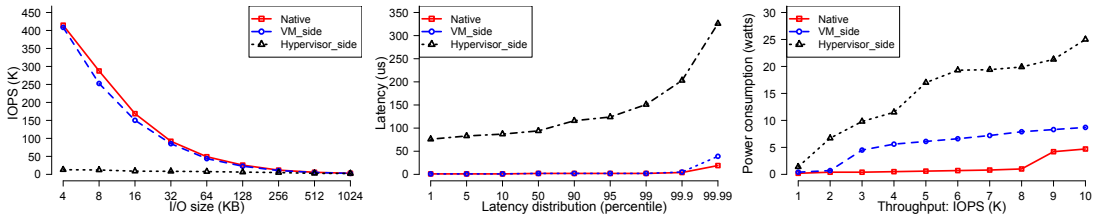
In the virtual I/O path, there are two operations that are expensive in CPU cycles or in request latencies. The first is virtual I/O emulation, which requires intensive interactions between the *virtio* frontend and backend. Emulation causes frequent I/O interrupts and guest I/O exits, which are expensive in CPU cycles, as well as increase I/O latency. The second is the relatively slow HDD-based storage access, which costs milliseconds and has long been the bottleneck of cloud applications.

Both VM-side and hypervisor-side cache hits avoid the HDD access, thus, obviate the HDD latency. For hypervisor-side caching, *virtio* and *qemu* are always in the I/O critical path, thus, I/O virtualization overheads are inevitable. In contrast, for VM-side caching, if it's DRAM-based cache, *virtio* and *qemu* are not involved in the

Fig. 28. The meter used for power consumption measurement.

I/O path, because the DRAM of a VM is managed by *KVM* instead of *qemu*. *KVM* is much more efficient than *qemu* userspace emulation, especially with the support of hardware-assisted virtualization. If the cache is PCIe SSD-based, and the cache device is allocated to the VM via *PCI passthrough*, the I/O virtualization layers are also obviated. *PCI passthrough* is supported by *IOMMU*, which enables direct remapping of the guest physical address to host physical address, thus avoids I/O virtualization layers to apply the translations and obviates the I/O operation delay. However, if it's SATA SSD-based, even if the cache is logically VM-side, since the access to the cache device needs the involvement of *virtio* and *qemu*, I/O virtualization overheads still exist. As a result, VM-side caching is not superior to hypervisor-side caching for SATA SSD devices. To understand the I/O virtualization penalties as well as the performance and energy efficiency characteristics of various cache deployments, we quantitatively compare different cache schemes. Insights from the evaluation can direct future cache designs and optimizations of virtualization systems.

To evaluate the virtual I/O overhead, we use *fio* [5] as the I/O benchmark. *fio* enables various I/O workloads with optional parameters including *read/write* type, *sequential/random* access, I/O size, IOPS, and *O_DIRECT* etc.. We run *fio* on VMs. Setting the *direct* parameter enables I/O requests bypassing VM-side caches and hit-



(a) Maximum throughputs with various I/O sizes.

(b) Latency distributions of 4KB *randread* at the IOPS of 5000.

(c) Power consumption of 4KB *randread* at various IOPS.

Fig. 29. *fio* benchmark on DRAM-based Caches. *Native* denotes *fio* directly runs on the host machine and hits the host OS page cache; *VM_side* denotes *fio* runs on the VM and hits the guest OS page cache; *Hypervisor_side* denotes *fio* runs on the VM, misses the guest OS page cache but hits the host OS page cache.

ting hypervisor-side caches. *fio* reports benchmark performance such as IOPS and latency distribution. We measure the power of the whole machine, because the cache access in virtualization platform involves intensive activities of multiple resources, including cache devices and CPUs. A *Watts Up? Pro ES* meter shown in Figure 28 is used to measure the wall power of the machine.

Our system is equipped with an AMD Phenom II X4 B95 Quad-core 3.0 GHz processor with AMD-V virtualization support. The host OS is a 64-bit Ubuntu 15.04 with Linux kernel version 3.19.0-30-generic. QEMU emulator version 2.4.1 and KVM are used as the hypervisor. An official Ubuntu 15.04 64-bit Server Cloud Image is run on the VM as the guest operating system with 2 VCPUs and 2GB memory.

We choose *random read* as the I/O pattern of the *fio* benchmark to minimize the interference caused by potential data prefetching of operating systems. For each cache setting, we report the maximum throughput, latency distributions, and energy consumption of the benchmark. For latency distribution, we focus on the I/O size of 4KB, which is the default page management unit of most Linux operating systems.

Observation 1 (on DRAM cache):

The performance of VM-side caches is very close to the native caches in throughput and per I/O response time; while the hypervisor-side caches have an up to 97% performance penalty.

Observation 2 (on DRAM cache):

For 4KB small I/O requests, the maximum throughput of hypervisor-side caches is only about 3% of the VM-side caches; for 1MB large I/O requests, the maximum throughput of hypervisor-side caches is nearly the same as the VM-side caches.

Observation 3 (on DRAM cache):

For same I/O throughput, hypervisor-side caches consume about 3x the power of VM-side caches.

As it's shown in Figure 29(a), with various I/O sizes, the VM-side cache consistently achieves near-native performance with a gap of less than 15%. In contrast, the hypervisor-side cache has a performance penalty of up to 97%. In Figure 29(b), the VM-side cache consistently achieves a near-native per I/O response time. In contrast, the hypervisor-side cache has a response time penalty of nearly 65 μs for a single 4KB read request. In Figure 29(c), for a same I/O throughput, the hypervisor-side cache consumes up to 3x the power of VM-side cache.

For DRAM, VM-side caches perform better and consumes less power than hypervisor-side caches. We believe the main reason is that the VM-side cache hit bypasses the *I/O virtualization layer*. Memory virtualization is implemented in the *KVM* kernel module, which is efficient with the support of hardware-assisted virtualization techniques such as *Intel VT-x* and *AMD-V*. In contrast, the I/O virtualization, including virtual I/O operations and disk emulation, is mainly managed by *QEMU*, which is a userspace process. The execution of virtual I/O requires frequent CPU mode switches, such as switches between *user* and *kernel* as well as *kernel* to *guest* mode, which are expensive in CPU cycles. As it's shown in Figure 27, when cache hits at *1a* (VM-side DRAM cache), virtual I/O and disk emulation are bypassed, the disk

Table 6. *perf* system-event statistics during a 30-second cache access period with an IOPS of 5k. We ensure the cache hit at VM-side and hypervisor-side, respectively. *fio* is running on the VM; *perf* is running on the host OS to probe the system events caused by the VM process. For a direct comparison, the percentages of system events are normalized to the total number of hypervisor-side events. The units of *Count* and *Percent* are *million* and *%*, respectively.

Event Source	Cache hit location			
	Hypervisor		VM	
	Count	Percent	Count	Percent
kernel	17545	60.90	1569	5.45
qemu	3295	11.44	0	0.00
kvm	2714	9.42	1478	5.13
kvm_amd	1811	6.29	1176	4.08
libglib	1220	4.23	0	0.00
libpthread	936	3.25	0	0.00
vdso	675	2.35	0	0.00
libc	609	2.12	0	0.00
Total	**28809**	**100**	**4225**	**14.66**

I/O operation is actually transformed to virtual memory access which is managed by *KVM*. When cache hits at *2a* (hypervisor-side DRAM cache), it implies a cache miss at *1a*, although the disk access can be avoided, the virtual I/O and disk emulation operations are still involved, thus, longer response time is observed by applications running on the VM, as well as a higher system power consumption.

To further verify our explanation, we conduct system event statistics during cache access under different caching schemes. *fio* benchmark is running inside a VM, and *perf* utility is employed to monitor the system events caused by the VM process. The *perf* statistic results are shown in Table 6. Generally, for a same amount of I/O requests, in hypervisor-side cache scheme, the total number of VM caused system events is 6x of the VM-side cache scheme. Specifically, in VM-side cache scheme, there are few userspace events caused, while, in hypervisor-side cache scheme, there are considerable user space system events such as events caused by *qemu* process and

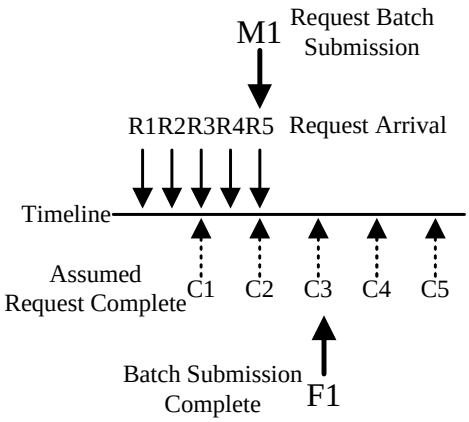


Fig. 30. Virtual I/O batch submission.

GLib library, etc.. This statistical analysis explains why hypervisor-side cache access is more costly than VM-side cache access.

From Figure 29(a) we can observe that as the I/O size increases, the throughput gap between *VM_side* and *hypervisor_side* cache schemes narrows down. We believe the reason is that for virtual I/O requests, the communication time between the *front-end (VM)* and the *back-end (Hypervisor)* is almost constant, thus for small requests the response time is dominated by the virtual I/O *round trip time (RTT)* between the VM and the hypervisor. When the request size increases, the real data transfer time dominates and the *RTT* becomes ignorable, thus, the throughput gap between *VM_side* and *hypervisor_side* cache narrows down.

Differing from Figure 29(a) and Figure 29(b) in which the *VM_side* and *Native* lines are almost overlapping, in Figure 29(c) there is an obvious gap between the *VM_side* and *Native* lines. The reason is that although VM-side memory access has a close-to-native performance, memory virtualization involves intensive activities of *KVM* module, which consumes extra power compared with native memory access.

4.1.3 Existing Virtual I/O Optimizations and Their Limitations

4.1.3.1 I/O batch submission

Virtual I/O batch submission has been proposed to enable SSDs achieving maximum throughput in virtualization environments [38]. I/O batch submission is an optimization for amortizing the I/O virtualization costs. The processing of a virtual I/O request requires the involvement of the userspace process such as *QEMU* as well as incurs *user-kernel* and *kernel-guest* CPU mode switches. Since these costs are per I/O based, I/O virtualization layer mainly limits the I/O request rate, instead of the data transfer rate. As a result, I/O batch submission can improve the overall system throughput. As an example, let's assume that 1) requests arrive evenly; 2) the request arrival rate is two times of the system processing rate; 3) every five adjacent requests are submitted in batch. As it's demonstrated in Figure 30, if batch submission is not employed, the fifth request *R5* will complete at time point *C5*. That's the five requests are completed in *C5* time. If batch submission is used, *R1* will not be served until the arrival of *R5*, and the five requests will complete at time point *F1*, which slightly lags *C3*. That's the the five requests are completed in *F1* time. Since *F1* leads *C5*, the system overall throughput is increased. However, batch submission has a potential problem, that's part of the requests may observe prolonged response time. For example, without batch submission, *R1* is assumed to complete at *C1*, but it's actually complete at *F1* which lags *C1* and make *R1* undergoes long response time. Another potential problem for I/O batch submission is that not all applications welcome asynchronous response. For web structure mining applications, the content of a page must be return before the subsequent read request could be made, since the link of next read request is contained in the response result of its predecessor request.

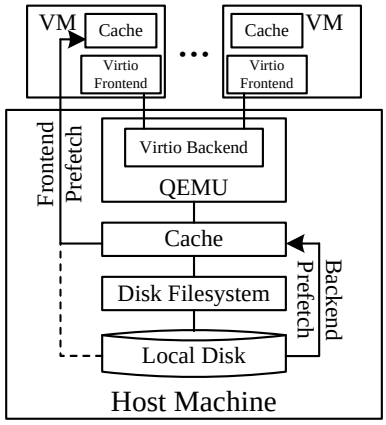


Fig. 31. Virtual I/O backend prefetching vs. frontend prefetching (our method *VIP*).

4.1.3.2 Virtual I/O backend prefetching

Traditional data prefetching focuses on applications running on bare metal systems using hard drives. SSDs have become popular alternatives over hard drives for data-intensive applications running on virtualized servers. Considering that prefetching in every VM may cause too much overhead and the I/O blending effect in virtualization storage stack, *VIO-prefetching* [19] implements a prefetcher in the host system with the guest I/O process identification as the hint to accurately recognize I/O patterns. More specifically, instead of only taking the block address into consideration to recognize I/O patterns, *VIO-prefetching* also utilizes the *PID* information to limit the pattern recognition in a separate I/O stream of an application so as to avoid the interference caused by disk sharing. As it's demonstrated in Figure 31, VIO-prefetching improves the virtual I/O performance since part of read requests originally target SSDs can be served by the faster host-side DRAM caches. However, the host-side prefetching does not bypass the costly virtual I/O operations, because the virtual I/O path is above the host-side cache. In other words, even data are promoted into the host-side cache by prefetching, access the host-side cache still involve

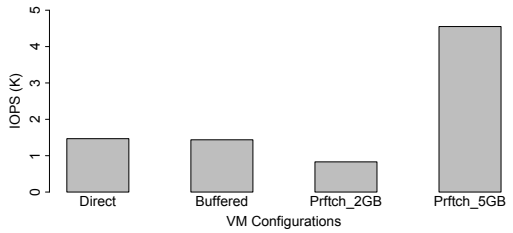


Fig. 32. The impact of virtual I/O front-end prefetching on VM storage performance of Google Compute Engine. *fio* benchmark runs in a VM, issuing 4KB *randread* requests on a 5GB file. *Direct* denotes *fio* requests bypass the VM-side cache and arrive the virtual block device; *Buffered* is similar to *Direct*, but the requested data will also be brought into the VM-side cache; *Prftch2_GB* denotes the VM *block device read ahead* is enabled, but the cache size is limited at 2GB. Similarly, *Prftch_5GB* denotes the cache size is set as 5GB.

the virtual I/O path, which is expensive in latencies and CPU cycles [51].

4.1.4 Optimizing Virtual I/O: Chances and Challenges

For large requests, instead of the communication overheads, data transfer dominates the virtual I/O response time. Thus, for workloads with intensive small requests, front-end prefetching, which has a chance to merge a batch of small requests into fewer larger ones, enables the communication overheads to be amortized. Even for the case that caches are built at the hypervisor side (*virtio* backend), the *virtio* frontend prefetching can further improve the VM storage performance and system energy efficiency. For instance, *block device read-ahead* of the guest OS can be set to prefetch data upon each block I/O request. The result is that for a 4KB virtual I/O request, a larger bulk of data such as 128 KB can be read ahead into the VM buffer. This will benefit subsequent read requests targeting prefetched pages. A similar method has been employed by network file systems such as *NFS*, in which the default block size is 1MB instead of 4KB, so as to avoid frequent network com-

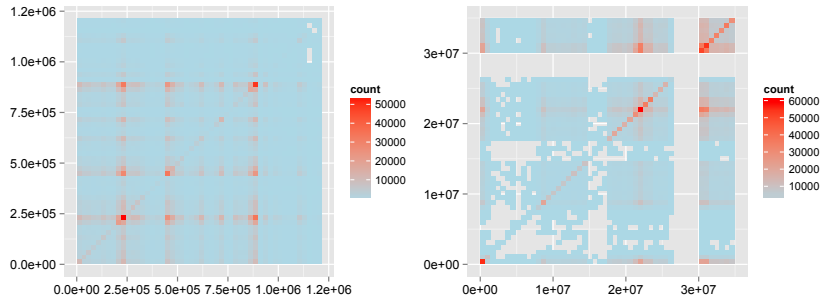
munication between clients and servers. Moreover, our tests on virtual I/O energy consumption with various I/O rates show that as the IOPS increases the virtual I/O caused power has a Logarithmic growth. For example, an IOPS of 3k causes the machine power increasing by 10w. While the IOPS is 6k and 9k, the power consumptions are 19w and 21w, respectively. This implies that a more intensive I/O rate has a lower per I/O energy consumption. Unlike on-demand I/O that an I/O request will only be issued upon its arrival, prefetching can predict which I/O requests will come and issue them in advance of their arrival. In other word, virtual I/O prefetching can shape the I/O issuing patterns. Thus, during the prefetching period in which the I/O request rate is high, the system will have a lower per I/O energy consumption than no prefetching is employed. As a result, virtual I/O prefetching may improve the system energy efficiency.

We conduct a group of tests on *n1-standard-1* VM instances of Google Computer Engine to evaluate the impact of front-end prefetching on VM storage performance. We utilize the default Linux OS *block device read ahead* mechanism as the prefetching implementation. From Figure 32 we have four observations. First, prefetching improves the VM I/O throughput. When the cache is 5GB, it can be guaranteed that all of the prefetched data will be reused before eviction, thus, it's an optimal case. From the figure we can see that the VM throughput increases by 200% when prefetching is enabled and the cache is large enough to ensure all prefetched data being reused. Second, prefetching imposes costs which are related to the prefetching granularity. The costs mainly come from two sources: read data from the disk and bring data into the cache. Comparing with direct I/O, which bypasses the cache and directly read disk, buffered I/O, which bring the read data into the cache, incurs 2% degradation of throughput. When prefetching becomes more aggressive, the prefetching overhead may also becomes more obvious. In the case that the prefetching cache size is set as

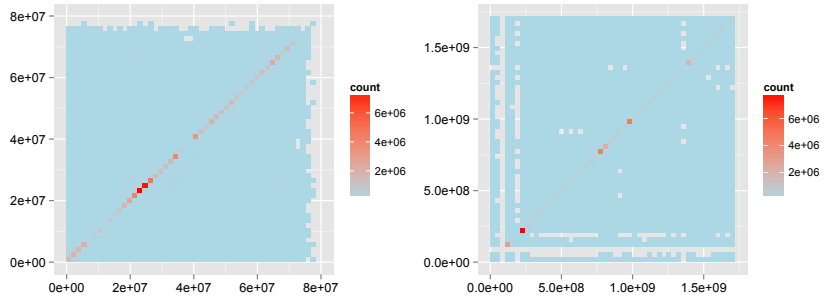
2GB, which cannot ensure all of the prefetched data being reused, if the default guest OS prefetching is enabled, the VM throughput degrades by more than 40%. Third, the size of prefetching buffer has important impact on the prefetching effects. Larger cache implies that data can reside longer in the cache before being flushed out, thus, higher possibility that prefetched data will be reused before eviction. Our test has shown the dramatic performance difference between the cache size of 2GB and 5GB.

The main challenges for efficient virtual I/O front-end prefetching are twofold. First, improving the prefetching accuracy is hard but important for VM storage performance. This is especially true when the size of the prefetching buffer is limited. As it's demonstrated, when the buffer size is limited, the native sequential prefetching may dramatically degrade the VM storage performance due to considerable fruitless but costly prefetches. Since the main memory is one of the most expensive resources in cloud platforms, less memory usage for prefetching buffer means less costs for tenants. This also makes an accurate prefetching crucial. Second, reducing the number of prefetching requests is important for improving system energy efficiency. Prefetching accuracy can improve VM storage performance, but the system energy consumption mainly depends on the I/O rate. If a sequence of I/O requests is prefetched at the granularity of per I/O, the total number of I/O requests will not be reduced, thus, we cannot expect improved system energy efficiency. For a given workload, it's important to merge the requests so as to reduce the number of prefetching requests as well as to improve system energy efficiency.

We conduct trace analysis to investigate the block I/O patterns of practical workloads. The insights can direct accurate prefetching for the design of the *VIP* algorithm.



(a) Umass F2 trace. (b) Umass Web3 trace.



(c) MSRC Prxy1 trace (d) MSRC Usr1 trace

Fig. 33. Bi-block correlations mined from block I/O traces. X axis represents a block address; Y axis represent the next address. The occurrence count of a sequence xy is demonstrated using various fill gradients. A red spot (x,y) in the figures implies a high occurrence frequency of the sequence xy. Here, x and y can be ranges instead of single points.

4.2 Trace Analysis

We investigate the block traces of various servers to identify temporal and spatial access features which can direct efficient virtual I/O prefetching. I/O predictability is important for the prefetching accuracy. For example, if I/O requests in a sequence are fully random without any patterns, it's impossible to achieve an accurate prefetching because upon the arrival of an I/O it's very hard to predict which I/O will follow it. Fortunately, block device access is usually not fully random, because the files or database records all contain semantic information which makes the accessing to

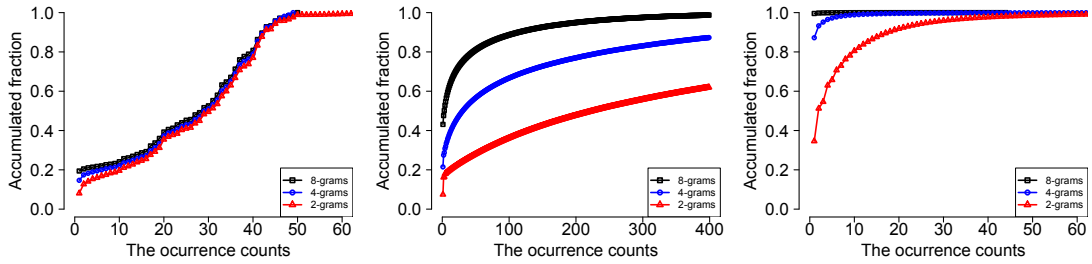
these data follow some patterns. For example, in OLTP systems, transactions may be conducted repeatedly and each transaction involves multiple records. Since the block positions of these records are relatively invariable, it's highly possible that the accessed block addresses will follow some patterns and these patterns are constant in a relatively long time. As a result, block device I/Os could be predictable.

Virtual I/O front-end prefetching mainly focuses on optimizing workloads dominated with small read requests. To choose the block I/O traces, we select workloads with the following features: read dominant, small I/O dominant, and I/O intensive. More specifically, we choose the workloads with a read ratio of more than 60%, an average read size of less than 16KB, and an average inter-arrival time of less than 15ms.

4.2.1 Spatial Correlation Behavior

Since stable patterns of block accesses are the basic premises for effective prefetching, we investigate the spatial correlation behavior of various workloads. We begin our trace analysis by asking the following question. *Given an LBA* $\boldsymbol{x}$*, is the subsequent LBA* $\boldsymbol{y}$ *statistically predictable?*

We first investigate the spatial correlation behavior of block I/O accesses. Figure 33 plots the block correlations recognized from UMASS Traces and MSR-C Traces [55]. We plot dual-block correlations in this figure. In a sequence of block accesses, we plot a corresponding point at `(x,y)` if block `x` is accessed instantly after block `y`. From the figure, we observe that there exists frequent dual-block sequences shown as dark areas in the figures. In all of these sub-figures, there are a few extremely hot spots, which imply stable and frequent access patterns. Given a block I/O address, the repetition of I/O patterns provides chances to predict its subsequent blocks for prefetching.



(a) Umass F2 trace. (b) MSRC Prxy1 trace. (c) MSRC Usr1 trace.

Fig. 34. CDFs of *8-gram* I/O sequence occurrence counts.

Trace Observation 1:

For an LBA `x` in an access sequence, its subsequent LBA `y` usually falls into a few LBA ranges. We call this observation *statistical predictability* of block accesses.

Trace Implication 1:

Statistical predictability of block accesses provides chances for effective prefetching.

4.2.2 Spatial Distances between Adjacent Block Accesses

As it's demonstrated in Figure 33, every sub-figure has a dark diagonal line, which implies spatial locality. Sequential prefetching is the default block prefetching mechanism of most operating systems. However, sequential locality is a special case of spatial locality. Even though a workload shows spatial locality, if it is not strongly sequential, traditional sequential prefetching may not be effective. We continue our trace analysis by investigating the following question. *Is sequential prefetching effective for production workloads?*

The dual-block correlation pattern of spatial locality can be represented as `x` $\rightarrow$

Table 7. Spatial distance (in blocks) of adjacent I/O accesses of the selected I/O workloads.

%th distance	Trace Repository					
	MSRC				UMASS	
	prxy1	**proj2**	**usr1**	**src1_1**	**F2**	**Web3**
10	32	32	128	128	6	64
20	128	104	128	264	36	96
30	384	128	256	11k	1898	317k
40	560	128	384	334k	4708	1108k
50	1024	176	384	1637k	8372	2439k
60	2136	256	640	4344k	16k	7313k
70	629k	512	896	9215k	201k	10600k
80	9850k	1024	1408	19913k	644k	13109k
90	26827k	89402k	3343k	45658k	654k	29830k
99	57240k	697659k	770137k	446271k	664k	33830k

$\mathtt{x} \pm \mathtt{k}$. MSR-C traces have strong spatial locality, while the UMASS traces have relatively weak spatial locality. Spatial locality potentially validates sequential prefetching. However, sequential prefetching is effective only if the value of `k` is small. Therefore, we further investigate the spatial distances between adjacent block accesses. Percentile distances between adjacent block accesses of each trace are listed in Table 7. For example, the 10th, 20th, and 50th percentile spatial distance between adjacent block accesses of *prxy1* workload is 32, 128, and 1024 blocks, respectively. It implies that sequential prefetch of 32 blocks only provides a 10% possibility making a subsequent block included in the prefetch sequence. To increase the possibility to 20% and 50%, the prefetch length needs to be 128 and 1024 blocks. The common default maximum *block read ahead* parameter of modern Linux operating systems is 256 blocks, which can provide a possibility of only about 25%, 60%, 30%, 20%, 20%, and 25% to make the prefetch useful for subsequent block I/O for *prxy1, proj2, usr1, src1_1, F2*, and *Web3* trace, respectively. Aggressive prefetch may increase the hit possibility, but it imposes high system pressure. Moreover, a same block may involve in multiple patterns. In figures such as Figure 33(a) and 33(b), an `x` point may correspond to more than one "hot" `y` points. The one-to-many mapping between `x` and `y` implies that a block element can be contained in multiple patterns, which can be

out of reach by sequential prefetching.

Trace Observation 2:

Spatial distances between most adjacent accesses can be relatively large; a single LBA may involve multiple access patterns.

Trace Implication 2:

Sequential prefetching is not effective; advanced prefetching algorithms are desired.

4.2.3 Recurrence of I/O Sequence Patterns

Spatial locality is an important hint for prefetching, but it's not the sole hint. Taking the OLTP application for example, a database `select` transaction may involve multiple records which need to be read in a logically consistent order from the disk but these records are not stored sequentially in the disk. In other word, for a dual-block access sequence `AB`, it's possible that `B` frequently follows `A` but `B` is not the in-disk neighbor of `A` or even close to `A`. In this case, spatial locality based sequential prefetching cannot make a correct decision to preload `B` into the cache upon the access of `A`.

The ineffectiveness of sequential prefetching in the above case is caused by the starkness of the prefetching algorithm, which only works for the specific sequential access mode. We finalize our trace analysis by investigating the following question. *Which access characteristics hidden in production workloads can be used to direct intelligent prefetching?*

Analyzing block I/O traces, we find that a lot of block access sequences recur frequently in the access history. We take a sequence of eight continuous I/O requests

as a unit, and count its occurrence in the access history. Figure 36 demonstrates that for *F2* trace only about 20% 8-gram sequences are only accessed once in a 10-hour period; more than 78% *8-gram* sequences are repeatedly accessed. Some sequences even recur up to 50 times. For *Prxy1* trace, about 42% *8-gram* sequences are only accessed once in a 1-week period; more than 58% sequences are repeatedly accessed. Some sequences even recur more than 400 times.

Trace Observation 3:

Ensembles of LBA sequences frequently recur in block I/O history.

Trace Implication 3:

Access history based prediction can potentially be effective for intelligent prefetching.

To predict the relation between the I/O characteristics of an application and its LBA sequence recurrence, a thumb rule is that if the spatial accesses of the application are skew such as 80% I/Os target 20% LBAs, the average I/O size is small such as several KBs, and the average I/O arrival rate is high such as hundreds per second, then the application may have frequent LBA sequence recurrence.

4.3 Virtual I/O Prefetching (VIP) Algorithm

Based on the insights we gain in Section 4.2, we design a virtual I/O prefetching algorithm. We take the block correlation recognition, prefetch buffer size, and buffer replacement algorithm into consideration. We use a prefetch simulator with different configurations to compare the cache hit ratio of sequential prefetching and our prefetching algorithm *VIP*. The goal is to investigate the effectiveness of our algorithm

for different workloads.

4.3.1 Recognizing Block Correlations

The rationale behind sequential prefetching is that if a data element is accessed, its next data element will locate within relatively close storage locations. This rationale can be represented as a block access association rule $\mathtt{x} \rightarrow \mathtt{x} \pm \mathtt{k}$, and $\mathtt{k}$ is a small number. However, our quantitative analysis on spatial distances between adjacent block accesses in Section 4.2.2 demonstrates that $\mathtt{k}$ is not always small enough for a sequential prefetch to take effects. Therefore, more advanced block correlation mining algorithms are desired. Fortunately, our analysis in Section 4.2.3 shows the chances for designing access history-aware block correlation mining algorithms.

A computer system event such as a file I/O operation can cause one or more block accesses. Assuming an I/O event `A` causes multiple block accesses `a1-a2-a3` which happen in a predefined order. Then, every time the event `A` happens, the block sequence `a1-a2-a3` recurs. As a result, in the block I/O access history, the block sequence `a1-a2-a3` repeatedly happens. We call this intra-event block correlation. Assuming I/O event `A` only accesses a single block `a` and event `B` accesses `b`. If there exists a strong correlation between `A` and `B`, it means that when `A` happens, B has a high possibility to happen. As a result, if `A` is a frequent event, the block sequence `a-b` will repeatedly happen in the block access history. We call this inter-event block correlation. In both cases, as far as a sequence repeatedly happens, a history-aware block correlation mining algorithm can recognize `a1-a2-a3` and `a-b` as frequent sequences, and discover $\mathtt{a1} \rightarrow \mathtt{a2}$ and $\mathtt{a} \rightarrow \mathtt{b}$ as association rules.

A storage system can be considered as a discrete-time dynamical system. The storage space with LBA set consists of a state space. The `arrival time` of a request and the `LBA` of the request correspond to `time` and `state` of the dynamical system.

Table 8. The conceptual correspondence between a storage system and a general discrete-time dynamical system.

Storage System	**Discrete-time Dynamical System**
LBA set	State space
LBA	State
Request arrival time	Time
Transitions from one LBA to another	Markov chain
Spatial locality	Clumping

The transitions from one LBA to another in the storage space can be modeled as a Markov chain. The spatial locality of a storage system can be seen as state space clumping in a dynamical system. The conceptual similarities between a storage system and a general discrete-time dynamical system are listed in Table 8.

Our trace analysis has demonstrated the visual predictability of one block I/O request to its subsequent request. Also, the block accesses of a storage system have the following characteristics:

(1) At a specific time point `t`, system is accessing block at `LBA_t`.

(2) State set `E`={`LBA_1, LBA_2, ...`} is finite.

(3) "`time`" is discrete.

(4) There are probabilistic transitions between states (`LBAs`).

(5) Morkov property: future `LBA_f` of the access process depends only upon the present state `LBA_p`.

Considering the storage access process perfectly fits the Markov process, we model the storage access using Markov chain. We observe that the transitions from one LBA to another are not strictly memoryless. For example, assuming there are two frequent sequences `a-b-c` and `b-d` in the block access history, if only knowing the present access is state `b`, the next state can be `c` or `d`. However, if knowing the preceded state was `a` and present state is `b`, then it's highly possible that the next state will be `c` instead of `d`. Therefore, we believe an *n-order* Markov chain is more accurate for modelling the block I/O process.

4.3.2 Core Algorithm

For a sequence of historical LBA accesses, *VIP* employs Markov chain to recognize the LBA correlations. The core algorithm mainly consists of 3 stages: (1) generating Markov chains for all unique LBAs; (2) indexing the Markov chains; and (3) recognizing LBA correlations based on the Markov chains with predefined *support*, *confidence* thresholds, and maximum correlated sub-sequence length parameters. The algorithm is described as follows.

Algorithm: `VIPCandidate(s,sup,conf,length)`

Input: LBA access sequence `s`,

support threshold `sup`,

confidence threshold `conf`,

maximum correlated sub-sequence length `length`.

Output: The correlated LBA sequences set `L`.

Procedure:

1: for each `k` in `[1...length]` do

MarkovChain[`k`] ← NGram`(s, n=k)`

2: for each unique `LBA` in `s` do

`seq` ← `LBA`

for each `k` in `[1...length]` do

`n` ← MarkovChain[`k`].search(`seq`)

if `support`(`seq` → `n`) ≥ `sup` and

`confidence`(`seq` → `n`) ≥ `conf` and

`k` < `length`

`seq` ← `seq` ⋄ `n`

else

add `seq` into `L`

continue

add `seq` into `L`

3: return `L`

Note: ⋄ denotes concatenating an LBA to a sequence.

In the first stage, the algorithm generates the *1*-order to *n*-order Markov chains

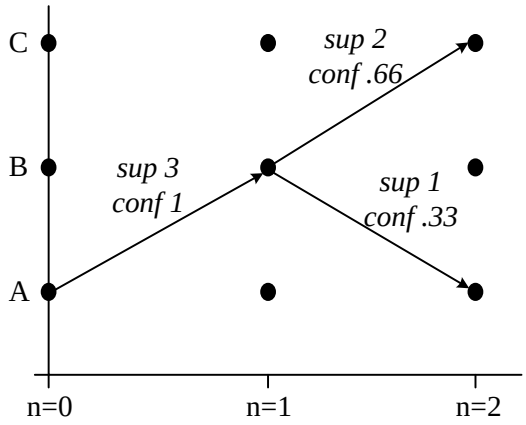


Fig. 35. Possible paths of one-step and two-step transitions from state `A` to other states for a three-state Markov chain.

of all unique LBAs. Here, n equals the predefined maximum correlated sub-sequence length. All of the generated Markov chains are indexed as *sorted sets* to accelerate searching. Currently, we use *Redis* in-memory key-value database as the storage engine of Markov chains. As an example, for a sequence {A B C A B A B C}, its *1-grams* and *2-grams* are:

A: B {3}

B: C {2} | A {1}

C: A {1}

A B: C {2} | A {1}

B C: A {1}

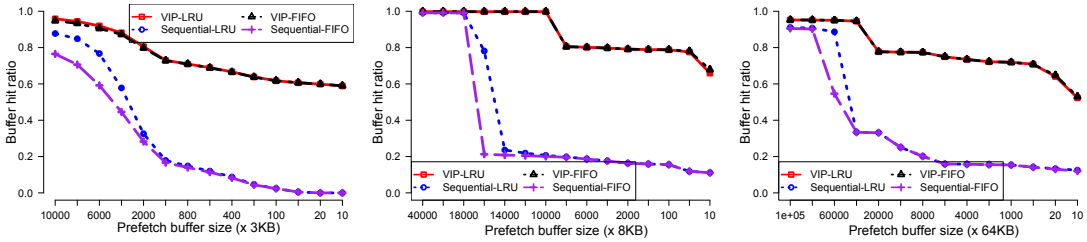
C A: B {1}

B A: B {1}

Future access prediction can be considered as a multi-step state transition from present state to a possible future state for a Markov chain. We can construct the

state transitions of the Markov chains in the above example as Figure 35. Upon a block I/O request, to predict its future two accesses *VIP* first searches the *1*-order Markov chains, also shown as *1-grams* and their possible subsequent elements, to gain the next LBA which has the highest possibility to be accessed. Then, with the two elements, the *2*-order Markov chains are searched to gain the third element. During this process, proper *support* and *confidence* thresholds can be used to make the results accurate. Considering the above sequence, if the present LBA is `A` with *support* and *confidence* thresholds of 2 and 0.5, the prediction results of the future two accesses will be the sequence {`B C`}.

For a sequence consists of L elements, constructing its *n-order* Markov chains has a time complexity of *O(L)*, which has very little optimization space. However, constructing *n-order* Markov chains has a spatial complexity of *O(nL)*. In other word, the memory consumed to save the Markov chains is proportional to the order of the chains. We observed that for workloads with large working set sizes, saving the Markov chains can consume GBs of space. We make an optimization and a tradeoff to save space. First, we observed that most n-grams only occur once in the access history. Since these n-grams are not informative for prefetching, pruning them from the result set will save considerable space without harming the prefetching accuracy. Second, we limit the maximum order of the Markov chains. Empirically, we construct the chains with a maximum order of eight. Although it's possible that an correlated sequence contains more than eight elements, this sequence will be split into multiple 8-grams. This will slightly increase the prefetch request counts, but will considerably decrease the space costs.



(a) Umass F2 trace. (b) MSRC Prxy1 trace (c) MSRC Usr1 trace

Fig. 36. Sequential prefetching vs. *VIP*. We replay traces using various combinations of prefetching algorithms and buffer replacement policies. *VIP* and *NAIVE* denote our prefetching and sequential prefetching, respectively. *LRU* and *FIFO* denote the buffer replacement policies.

4.3.3 Simulation Description

We replay block I/O traces using a prefetch buffer simulator with the *sequential* prefetching and our *VIP* prefetching algorithms. We use FIFO and LRU as buffer replacement policies for each case. So we can comprehensively measure the performance of each one out of four combinations of prefetching algorithms and buffer management policies. We split each trace into two halves. *Sequential* prefetching only replay the second half with fixed prefetching rules. That's upon the access of `LBA_A`, data located from `LBA_A+1` to `LBA_A+n` will be prefetched. *VIP* block correlation recognition algorithm uses on the first half of a trace to generate rules, which are used to guide prefetching when replaying the second half of the trace. Upon the access of `LBA_A`, the block correlation algorithm of *VIP* will recognize an LBA sequence which includes the LBAs correlated to `LBA_A` with support and confidence values higher than the predefined thresholds.

4.3.4 Result Analysis

Based on trace simulation we have the following findings.

Simulation Finding 1:

VIP improves the buffer hit ratio by up to 73% compared to the commonly used *sequential* prefetching scheme.

We first compare buffer hit ratios of *VIP* prefetching and sequential prefetching. Our tests demonstrate that VIP always performs better than the sequential prefetching and improves the buffer hit ratio by up to 73%. When the buffer size is relatively large, for example hundreds of MB for MSR-C Prxy1 trace and several GB for MSR-C Usr1 trace, sequential prefetching may also have a good cache hit ratio. However, in the case that VM DRAM is a scarce resource and the available buffer size is as small as several MB, VIP demonstrates huge comparative advantages over sequential prefetching.

Simulation Finding 2:

VIP achieves high buffer hit ratios even with extremely small buffer sizes.

Even when the buffer size is only 1MB, VIP can achieve a cache hit ratio of up to 75% for MSR-C Prxy1 workloads. In comparison, sequential prefetching can only achieve a hit ratio of 18% with an 1MB buffer size. Even with tiny buffer sizes of 30KB, 80KB, and 640KB for F2, Prxy1, and Usr1 workload, respectively, VIP can achieve a cache hit ratio of 60%, 64%, and 52% accordingly. In comparison, the hit ratios of sequential prefetching are 0, 12%, and 13%, respectively. So *VIP* achieves high buffer hit ratios even with extremely small buffer sizes. Assuming the DRAM space consumed by *VIP* is taken from the cache space, *VIP* can ensure a high buffer hit ratio with ignorable impact on the cache hit ratio.

Simulation Finding 3:

The effectiveness of *VIP* doesn't depend on the buffer replacement policies.

The hit ratios of traditional buffers or caches depend on the replacement policies. FIFO and LRU are two most widely used cache replacement policies. We evaluate the effectiveness of VIP and sequential prefetching under FIFO and LRU cache replacement policies. Tests show that the hit ratio of sequential prefetching depends on cache replacement policies. In all the three cases, LRU performs better than FIFO in sequential prefetching. But for VIP, FIFO and LRU performs similarly. So VIP is independent from cache replacement policies, which can potentially make the cache management simple.

Our simulation results verify that VIP can improve the cache hit ratio by more than 70% with a cache size of tens of MB. The average I/O throughput increases by more than 3x. VIP requires ignorable cache space to achieve high hit ratio. Since the virtual I/O cost is per I/O based, reducing the total number of I/O requests is crucial for high energy efficiency. Current I/O framework lacks the interface to merge non-sequential I/O requests. New prefetching interfaces are required to achieve efficient I/O merging and as a result virtual I/O energy efficiency. To implement new I/O prefetch interfaces and integrate VIP framework in real systems is our future work.

CHAPTER 5

RELATED WORK

In this chapter, we discuss systems, mechanisms, and other research efforts that are related to this dissertation. First, we survey VM migration including persistent storage. Then, we discuss host-side caching and the impact of runtime context changing on cache performance. Finally, we present various research efforts for QoS-aware storage management in cloud environments.

5.1 VM migration including persistent storage

Migration consumes considerable CPU, memory, and network resources. As a result, migrating a VM can slow down other collocated VMs [45]. If disk storage needs to be migrated, intensive read I/O incurred by migration will cause I/O contention. Resources like disk and network I/O bandwidth are not strictly partitionable, due to the nature of the hardware resource, or the limitations of current resource partitioning mechanisms. Consequently, resource contention may lead to unexpected performance fluctuation [42]. Migrating a VM including the local storage across WAN usually takes tens of minutes, if a large group of VMs are migrated, the period during which applications experience severe performance degradation is prolonged.

Given the complexity and variability of WAN, it's challenging to perform VM migration outside a local data center. The requirements of wide area migration differ greatly from that of LAN migration [20], the storage back-end associated with the VMs must also be migrated in wide area migration scenarios [1, 73, 59]. As the storage is the elephant in the data center, the storage migration will dominate the latency

of VM migration across WAN. To shorten the total migration time, data deduplication based methods are employed to mitigate transferring duplicated blocks among images [1, 73, 59]. The methods proposed in VMFlock, CloudNet, and Shrinker [1, 73, 59] reduce size of the data migrated, as a result shorten the total migration time considerably. However, they don't take the affinity among VMs into consideration during the live migration of a group of VMs. *Clique Migration* can be used as an optimization component and incorporated into the migration manager of these systems.

To meet the QoS of applications, instead of migrating the VMs one by one, LIME [36] migrates the network, VMs, and the management system as an ensemble. However, large scale virtual machine clusters contain thousands of VM nodes, the limited bandwidth of the WAN link may not be able to support migrating all of these VMs in parallel. Pacer [78] aims at synchronizing the migration progress of multiple VMs with dependencies. Pacer coordinates the migration of multiple VMs based on a prediction model, and dynamically allocates bandwidth as necessary. As a result, all the VMs complete migrating at nearly the same time. Coordinated migration can avoid the segmentation of application components over distant data centers, alleviating unacceptable application performance degradation. Pacer assumes each VM holds a single application component, and all components of an application will be migrated in parallel. In practice, an application may contain thousands of VMs, it's impractical to migrate all these VMs simultaneously. Both LIME and Pacer don't propose a grouping mechanism for further partitioning the related VMs. COMMA [77], on the other hand, empirically proposed algorithms to direct grouping and migration.

Starling [64], and AAGA [18] employ affinity-aware migration and grouping mechanism to minimize communication overhead in virtualized computing platforms, so as to improve the application performance. Meng's work [54] aims to improve the

scalability of data center network via optimizing the VM placement.

5.2 Caching in Virtualization Environments

Host-side caches have important impacts on the performance of virtualization systems. Various optimizations have been proposed. Eviction-based cache placement [35], hypervisor exclusive cache [48], and page deduplication [62] are proposed to mitigate memory waste of double caching. Considering the over-provisioning of DRAM resource in some systems, Mortar [31] implements spare memory pooling and global content-based shared cache, respectively, to fully utilize the idle memory resource and improve the VM storage performance. As server flash cache solutions are gaining adoption, flash-based caches are integrated into virtualization systems such as vSphere [70] and Mercury [15]. Write policies for host-side flash caches [41] are proposed to enable write-back on the host-side flash cache. Flash cache management mechanisms such as S-CAVE [52], client-side flash caching [4], vCacheShare [53], Centaur [40], and FVP [10] are implemented to maximize cache utilization, control VM storage performance, or tolerate host and flash failures. All these projects shed light on the importance of host-side caches.

The cold snap of caches that happen in new cache creation, running entities restart, and runtime context switching scenarios degrades application performance dramatically due to the plunge of the cache hit ratio. Cache warm-up is a process during which hot data are preloaded, usually in an aggressive manner, into caches to reduce access latency of succedent read requests. Restoring micro-architectural CPU-level caches in thread migration and core switch scenarios is discussed in [34, 14, 23, 74]. Restoring disk caches in new cache creation and server restart scenarios is discussed in Bonfire [75].

VM migration activities are considerable in production datacenters [12]. VM

migration [20, 56, 13] doesn't transfer host-side cache state, thus changes VM runtime contexts and degrades VM storage performance. Cache pooling [31, 10] is a coldness-avoidance mechanism, since after migration, hot data of a VM still reside in the cache of a remote machine. However, cache pooling commonly employs remote paging, which is not as efficient as accessing local caches due to network latency. Migrating the host-side cache [70] is another method to maintain the cache warmness after migration. However, it prolongs the total VM migration time. Bonfire-like mechanisms can be employed to warm up host-side caches in VM migration contexts. Nevertheless, VMs will undergo ultra-low storage performance during *Bonfire* cache warm-up period, which can be tens of minutes when the cache is tens of GB. Compared with these existing work, our work *Successor* aims to achieve a generic cache warm-up solution in VM migration contexts.

5.3 Virtual I/O Prefetching

Host-side caches have important impacts on the performance of virtualization systems. Various optimizations have been proposed. Eviction-based cache placement [35], hypervisor exclusive cache [48], and page deduplication [62] are proposed to mitigate memory waste of double caching. Considering the over-provisioning of DRAM resource in some systems, Mortar [31] implements spare memory pooling and global content-based shared cache, respectively, to fully utilize the idle memory resource and improve the VM storage performance. As server flash cache solutions are gaining adoption, flash-based caches are integrated into virtualization systems such as vSphere [70] and Mercury [15]. Write policies for host-side flash caches [41] are proposed to enable write-back on the host-side flash cache. Flash cache management mechanisms such as S-CAVE [52], client-side flash caching [4], vCacheShare [53], Centaur [40], and FVP [10] are implemented to maximize cache utilization, control VM

storage performance, or tolerate host and flash failures. All these projects shed light on the importance of host-side caches.

Comparing with accessing the local hard disks or remote network storage, host-side caching improves the VM storage performance. Host-side caches, however, are under virtual I/O back-ends. As a result, accessing the host-side cache involves the complete virtual I/O path, which is expensive in RTT and CPU cycles. Linux kernel community continuously optimizes the *virtio* drivers. Virtio-blk [27], Virtio-blk-data-plane [30], and Virtio-blk Multi-queue [43] have been successively implemented to improve the *virtio* performance. DID [68] was proposed to reduce the I/O virtualization caused interrupt delivery overheads so as to improve the virtual I/O performance. Study on *virtio* [60] with network transactions shows that a busy virtualized web-server may consume 40% more energy, due to 5x more CPU cycles to deliver a packet, than its non-virtualized counterparts [63]. Adaptive packet buffering is employed for processing packets in batch to amortize the interrupt cost and CPU time, so as to reduce energy consumption. To mitigate the performance degradation of multi-queue SSDs under virtual I/O, Kim et. al. [38] also implemented I/O batch submission via modifying the *virtio* [60] frontend and backend drivers. I/O batch submission is efficient for amortizing the overheads of I/O virtualization and improving system throughput and energy efficiency. However, employing batch submission to serve live requests will considerably prolong the response time of part of the requests due to the relatively long polling interval. To avoid requests undergoing long latency, instead of employing batch submission to serve live requests, our work *VIP* uses batch submission to serve prefetching requests, which are not latency sensitive but enable system to gain performance and energy efficiency benefits as well.

Data prefetching is widely used to reduce storage access latency. To achieve

optimal design, data prefetching needs to consider the performance characteristics of underlying storage devices. *Flashy prefetching* [69] harnesses the high bandwidth and salient random access characteristics of SSDs to enable prefetching for multiple simultaneous accesses. *ASP* [6] provides methods that resolve both independency loss and parallelism loss due to prefetching that may arise in striped disk arrays. *VIO-prefetching* [19] improves the virtual I/O performance through prefetching. However, *VIO-prefetching* adopts virtual I/O backend prefetching, which is conducted inside virtualization hosts, and mainly focuses on fully utilizing the SSD bandwidth to support prefetching. We recognize that the performance of virtual block devices is not only decided by the underlying storage media, but also limited by the virtual I/O driver, which conflicts with small requests. Being different from the existing work, our work *VIP* suggests implementing prefetching at the virtual I/O front-end so as to avoid the I/O virtualization overheads for cache access.

An FIFO history buffer, which naturally contains the page access sequences as well as the freshness of pages, has been used to hold the most recent miss addresses of main memory for effective data cache prefetching [57]. I/O access patterns and block correlations have been investigated to improve the effectiveness of storage cache prefetching [44, 19]. The key hint for correlation based prefetching is an address sequence, which is recognized to direct prefetching. In each prefetching, blocks contained in the sequence will be read ahead into the cache with the hope that they will serve and accelerate future requests. *VIP* adopts the similar idea to conduct prefetching. Current VIP algorithm is Markov chain based, and we have demonstrates its effectiveness.

CHAPTER 6

FUTURE WORK AND CONCLUSIONS

Fully exploiting the flexibilities brought by server virtualization ease the resource management and infrastructure maintenance of cloud platforms. However, it's important to mitigate the negative side-effects caused by virtualization features such VM mobility and I/O virtualization.

We started by investigating the impact of storage migration on virtual machine cluster performance. We focused on measuring the VM affinity in terms of network traffic. We found that the grouping and order in which VMs are migrated hugely affect cluster performance during migration. However, previous works don't take the VM affinity into consideration when making migration policies. Therefore, we presented our solution *Clique Migration*, a new VM migration mechanism that exploits inter-VM network traffic to partition VMs into subgroup, migrate VMs in same subgroups in parallel, to minimize the wide area communication traffic during migration so as to maximally maintain the virtual machine cluster performance during migration. We also found that migration degrades the performance of VMs which have host-side caching deployed. Therefore, we propose *Successor*, a new data prefetch mechanism to accelerate the host-side cache warm-up in migration contexts. I/O virtualization incurs additional overheads. Our tests on KVM virtualization platforms show that the virtual I/O sub-path adds an additional latency of about 60 μs. As high-performance NVM devices such as phase change memory (PCM) emerge, the overheads of virtual I/O become unacceptable. Finally, We propose VIP, an adaptive virtual I/O front-end prefetching mechanism for avoiding the frequent involvement of

virtual I/O stacks.

In this chapter, we first summarize the contributions of this dissertation. We then discuss several possible future research directions.

6.1 Summary

This dissertation is mainly comprised of two parts. In the first part, we investigated the impact of VM migration on VM system performance. We devised mechanisms to mitigate the impact and optimize system performance in VM migration contexts. In the second part, we fully exploited the software-defined performance feature of virtualization platforms to build a I/O tuning framework, which enables elastic storage volume performance in multi-tenancy environments.

6.1.1 Mitigating the Impact of VM Migration on System Performance

Affinity is common among Virtual Machines (VMs) in cloud environments. If VMs collaborating on a job are split in geographically distributed clouds, the low bandwidth and high latency inter-cloud communication via a wide area network (WAN) will dramatically degrade the system performance. A potential solution is migrating all of the VMs collaborating on a job in parallel, so as to avoid wide area communication. However, if the job is too large, it becomes impractical to migrate all of the VMs simultaneously due to limited WAN bandwidth and high block dirty rate. We present a migration optimization mechanism called Clique Migration for inter-cloud VM migration. First, the traffic monitor collects the traffic information between VM pairs. Then, the grouping mechanism profiles the traffic affinities, and makes a migration decision to decide which VMs should be migrated simultaneously, as well as the order in which each subgroup of VMs should be migrated. Analysis of the traffic trace of 68 VMs in an IBM production cluster shows that *Clique Migration*

can reduce inter-cloud traffic by 25% to 60%, when the degree of parallel migration is from 2 to 32. Tests of MPI multi-PingPing benchmark running on simulated inter-cloud environments, show that *Clique Migration* can significantly shorten the period during which applications undergo performance degradation. Tests of MPI Reduce scatter benchmark show that R-Min-Cut can keep the performance during migration at 26% to 75% of the non-migration scenario. Our tests also show that a synchronization mechanism like Pacer [78] is necessary in the migration manager to avoid wide area communication of VMs in the same subgroups caused by the out of step parallel migration.

In virtualization platforms, host-side storage caches can serve virtual machines (VM) disk I/O requests, which originally target network storage servers. When these requests hit host-side caches, network and disk access latencies are obviated, and thus VMs perceive improved storage performance. However, VM migration does not transfer host-side cache states. As a result, a newly migrated VM suffers performance degradation until the cache is fully rebuilt. The performance degradation period can be hours long if the cache is *naturally* warmed up. Employing existing cache warm-up solutions such as *migrating host-side cache* and *Bonfire*, VMs may either have a prolonged total migration time or undergo a performance degradation period of tens of minutes due to the warm-up caused storage contention. We present a host-side cache warm-up mechanism called *Successor*, which parallelizes cache warm-up and VM migration to minimize VM-perceived performance degradation period. *Successor* front-end snapshots cache state and logs VFS layer I/O requests. Combining this information with the size of destination cache and the available warm-up time, a candidate page set can be decided and used for the warm-up to maximize post-migration VM storage performance. Tests on the QEMU/KVM virtualization platform demonstrate that *Successor* can achieve optimal host-side cache warm-up in

storage migration scenario. In unplanned memory-only migration scenario, achieving optimal cache warm-up requires a longer migration time than warm-up time. I/O log-based cache alignment is a best effort optimization if the prerequisites of optimal cache warm-up cannot be met. Tests using *Zipf* workloads show that cache alignment can reduce the VM storage performance penalty. Through replaying IaaS VM traces, we verify the benefits of *Successor* for production clouds. The overhead analysis shows that *Successor* incurs less than 2% performance degradation to co-locating applications.

6.1.2 The System-level Optimization on Virtual I/O

In virtualization environments, hypervisor-side caches are deployed to improve storage system performance. However, hypervisor-side caches locate at the lower layer of VM disk filesystem, thus I/O virtualization stacks are still involved in the cache access critical path. Virtual I/O is expensive in terms of round-trip time (RTT) of the front-end and the the back-end, as well as CPU cycles, which cap the the throughput (IOPS) of hypervisor-side caches and incurs additional energy consumption [51].

If VM I/O requests are served by VM-side DRAM caches, the requests do not need go through I/O virtualization stacks, thus high I/O throughputs and low latency can be achieved. A challenge to implement an efficient front-end cache is bringing disk data into the cache in an efficient way. Our tests demonstrate that traditional sequential prefetching is not efficient for prefetch if I/O patterns are not sequential. Our trace analysis shows that block accesses are statistically predictable, which provides chances for effective prefetching. Also, our traces analysis shows that spatial distances between adjacent accesses are relatively long and a single LBA may involve multiple access patterns. Therefore, sequential prefetching is not effective and advanced prefetching algorithms are desired. Another observation is ensembles of LBA

sequences frequently recur in block I/O history. So access history based prediction can be effective for intelligent prefetching.

We propose virtual I/O front-end prefetching algorithm *VIP*, which employs Markov chains to recognize the LBA correlations. VIP generate Markov chains for all unique LBAs, index the Markov chains, and recognize LBA correlations based on the Markov chains with predefined *support*, confidence thresholds. The LBA correlations are utilized for cache prefetching. Our trace based simulation demonstrates that VIP improves the cache hit ratio by up to 73% compared with the widely used sequential prefetching. The high hit ration of VIP can be achieved even with less than 1MB cache space. And the effectiveness of VIP is independent from cache replacement policies.

6.2 Future Work

Virtualization promotes hardware resource utilizations and brings flexibilities to resource management and infrastructure maintenance in data centers. As we have pointed out, these benefits are not obtained for free. From the viewpoint of infrastructure management, VM migration changes the machine runtime contexts, thus degrades VM performance. We initiated research topics including VM grouping [49] for optimizing cluster performance in wide-area migration contexts and cache warm-up [50] in both wide-area and local-area contexts. Further research are expected to cover more contexts.

Virtualization adds additional layers to system software stacks. For example, due to the existence of both guest and host operating systems, the storage and network stacks become deeper. I/O forwarding has to go through more hops, thus higher latencies are common for I/O requests in virtualization environments. Our study on the impact of cache locations on storage performance and energy consumption

of virtualization systems [51] has verified the huge impact of virtualization layers on storage cache performance. Although reducing virtualization overheads is not a new topic, there remains huge unexploited research space.

BIBLIOGRAPHY

[1] Samer Al-Kiswany et al. "VMFlock: Virtual Machine Co-Migration for the Cloud". In: *HPDC'11*. San Jose, USA, June 2011.

[2] Mansoor Alicherry and T.V. Lakshman. "Optimizing data access latencies in cloud systems by intelligent virtual machine placement". In: *INFOCOM'13*. Turing, Italy, Apr. 2013.

[3] Michael Armbrust et al. *Above the Clouds: A Berkeley View of Cloud Computin*. Tech. rep. UCB/EECS-2009-28. UC Berkeley Technical Report, Feb. 2009.

[4] Dulcardo Arteaga and Ming Zhao. "Client-side Flash Caching for Cloud Systems". In: *SYSTOR'14*. Haifa, Israel, June 2014.

[5] Jens Axboe and Aaron Carroll. *fio(1) - Linux man page*. URL: `http://linux.die.net/man/1/fio`.

[6] Sung Hoon Baek and Kyu Ho Park. "Prefetching with Adaptive Cache Culling for Striped Disk Arrays". In: *ATC'08*. Boston, USA, June 2008.

[7] Paul Barham et al. "Xen and the Art of Virtualization". In: *SOSP '03*. Bolton Landing, NY, USA, Oct. 2003.

[8] F. Bellard. "QEMU, a Fast and Portable Dynamic Translator". In: *ATC'05*. CA, USA, Apr. 2005.

[9] Anton Beloglazov and Rajkumar Buyya. "Optimal Online Deterministic Algorithms and Adaptive Heuristics for Energy and Performance Efficient Dynamic Consolidation of Virtual Machines in Cloud Data Centers". In: *Concurrency and Computation: Practice and Experience* 24 (13 2012).

[10] Deepavali Bhagwat et al. “A Practical Implementation of Clustered Fault Tolerant Write Acceleration in a Virtualized Environment”. In: *FAST’15*. San Clara, USA, Feb. 2015.

[11] Robert Birke et al. “(Big)Data in a Virtualized World: Volume, Velocity, and Variety in Cloud Datacenters”. In: *FAST’14*. San Clara, USA, Feb. 2014.

[12] Robert Birke et al. “When Virtual Meets Physical at the Edge: A Field Study on Datacenters Virtual Traffic”. In: *SIGMETRICS’15*. Portland, USA, June 2015.

[13] Robert Bradford et al. “Live Wide-Area Migration of Virtual Machines Including Local Persistent State”. In: *VEE’07*. CA, USA, June 2007.

[14] Jeffery A. Brown, Leo Porter, and Dean M. Tullsen. “Fast Thread Migration via Cache Working Set Prediction”. In: *HPCA’11*. San Antonio, USA, Feb. 2011.

[15] Steve Byan et al. “Mercury: Host-side Flash Caching for the Data Center”. In: *MSST’12*. Pacific Grove, USA, Apr. 2012.

[16] Michael Cardosa et al. “Exploring MapReduce Efficiency with Highly-Distributed Data”. In: *MapReduce’11*. CA, USA, May 2011.

[17] Feng Chen, David Koufaty, and Xiaodong Zhang. “Hystor: Making the Best Use of Solid State Drives in High Performance Storage Systems”. In: *ICS’11*. Tuscon, USA, May 2011.

[18] Jianhai Chen et al. “AAGA: Afnity-Aware Grouping for Allocation of Virtual Machines”. In: *AINA’13*. Barcelona, Spain, Mar. 2013.

[19] Ron C. Chiang, Ahsen J. Uppal, and H. Howie Huang. "An Adaptive IO Prefetching Approach for Virtualized Data Centers". In: *IEEE Transactions on Services Computing* PP (99 2015).

[20] Christopher Clark et al. "Live Migration of Virtual Machines". In: *NSDI'05*. CA, USA, July 2005.

[21] Brian F. Cooper et al. "Benchmarking Cloud Serving Systems with YCSB". In: *SoCC'10*. Indianapolis, USA, June 2010.

[22] G. Crump. *VMware Server Side Caching - Flash SSD or DRAM*. 2013. URL: http://www.storage-switzerland.com/Articles/Entries/2013/6/18.

[23] David Daly and Harold W. Cain. "Cache Restoration for Highly Partitioned Virtualized Systems". In: *HPCA'11*. San Antonio, USA, Feb. 2011.

[24] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplied Data Processing on Large Clusters". In: *OSDI'04*. CA, USA, Dec. 2004.

[25] Tian Guo, Upendra Sharma, and Timothy Wood. "Seagull: Intelligent Cloud Bursting for Enterprise Applications". In: *USENIX ATC'12*. Boston, June 2012.

[26] Md E. Haque et al. "Few-to-Many: Incremental Parallelism for Reducing Tail Latency in Interactive Services". In: *ASPLOS'15*. Istanbul, Turkey, Mar. 2015.

[27] Asias He. *Virtio-blk Performance Improvement*. 2012. URL: http://www.linux-kvm.org/images/f/f9.

[28] David Howells. "FS-Cache: A Network Filesystem Caching Facility". In: *OLS'06*. Ottawa, CA, July 2006.

[29] Wenjin Hu et al. "A Quantitative Study of Virtual Machine Live Migration". In: *CAC'13*. Miami, USA, Aug. 2013.

[30] Khoa Huynh and Andrew Theurer. *KVM Virtualized I/O Performance*. 2013. URL: http://www.novell.com/docrep/2013/05.

[31] Jinho Hwang et al. "Mortar: Filling the Gaps in Data Center Memory". In: *VEE'14*. Salt Lake City, USA, Mar. 2014.

[32] *Intel MPI Benchmarks 3.2.4*. 2013. URL: http://software.intel.com/en-us/articles/intel-mpi-benchmarks.

[33] Sitaram Iyer, Antony Rowstron, and Peter Druschel. "Squirrel: a decentralized peer-to-peer web cache". In: *PODC'02*. NY, USA, July 2002.

[34] Nagakishore Jammula et al. "Balancing Context Switch Penalty and Response Time with Elastic Time Slicing". In: *HiPC'14*. Goa, India, Dec. 2014.

[35] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. "Geiger: Monitoring the Buffer Cache in a Virtual Machine Environment". In: *ASPLOS'06*. San Jose, USA, Oct. 2006.

[36] Eric Keller et al. "Live Migration of an Entire Network (and its Hosts)". In: *Hotnets'12*. WA, USA, Oct. 2012.

[37] Tae Yong Kim et al. "Improving Performance by Bridging the Semantic Gap between Multi-queue SSD and I / O Virtualization Framework". In: (May 2015).

[38] Tae Yong Kim et al. "Improving Performance by Bridging the Semantic Gap between Multi-queue SSD and I/O Virtualization Framework". In: *MSST'15*. Santa Clara, USA, Apr. 2015.

[39] Avi Kivity et al. "kvm: the Linux Virtual Machine Monitor". In: *OLS'07*. Ottawa, Canada, June 2007.

[40] Ricardo Koller, Ali Jose Mashtizadeh, and Raju Rangaswami. "Centaur: Host-side SSD Caching for Storage Performance Control". In: *ICAC'15*. Grenoble, France, July 2015.

[41] Ricardo Koller et al. "Write Policies for Host-side Flash Caches". In: *FAST'13*. San Jose, USA, Feb. 2015.

[42] Sajib Kundu et al. "Application performance modeling in a virtualized environment". In: *HPCA'10*. Bangalore, India, Jan. 2010.

[43] Ming Lei. *Virtio-blk Multi-queue Conversion and QEMU Optimization*. 2014. URL: `http://www.linux-kvm.org/images/6/63/02x06a-VirtioBlk.pdf`.

[44] Zhenmin Li et al. "C-Miner: Mining Block Correlations in Storage Systems". In: *FAST'04*. San Francisco, USA, Mar. 2004.

[45] Seung-Hwan Lim et al. "Migration, Assignment, and Scheduling of Jobs in Virtualized Environment". In: *Hotcloud'11*. Portland, USA, June 2011.

[46] Haikun Liu et al. "Performance and energy modeling for live migration of virtual machines". In: *HPDC'11*. San Jose, USA, June 2011.

[47] S. Lloyd. "Least square quantization in PCM". In: *IEEE Transactions on Information Theory* (28 1982), pp. 128–137.

[48] Pin Lu and Kai Shen. "Virtual Machine Memory Access Tracing With Hypervisor Exclusive Cache". In: *ATC'07*. San Clara, USA, June 2007.

[49] Tao Lu et al. "Clique Migration: Affinity Grouping of Virtual Machines for Inter-Cloud Live Migration". In: *NAS'14*. Tianjin, China, Aug. 2014.

[50] Tao Lu et al. "Successor: Proactive Cache Warm-up of Destination Hosts in Virtual Machine Migration Contexts". In: *INFOCOM'16*. San Francisco, USA, 2016.

[51] Tao Lu et al. “Understanding the Impact of Cache Locations on Storage Performance and Energy Consumption of Virtualization Systems”. In: *CoolDC’16*. Santa Clara, USA, Mar. 2016.

[52] Tian Luo et al. “S-CAVE: Effective SSD Caching to Improve Virtual Machine Storage Performance”. In: *PACT’13*. London, England, Sept. 2013.

[53] F. Meng et al. “vCacheShare: Automated Server Flash Cache Space Management in a Virtualization Environment”. In: *ATC’14*. Philadelphia, USA, June 2014.

[54] Xiaoqiao Meng, Vasileios Pappas, and Li Zhang. “Improving the Scalability of Data Center Networks with Traffic-aware Virtual Machine Placement”. In: *INFOCOM’10*. San Diego, USA, Mar. 2010.

[55] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. “Write Off-Loading: Practical Power Management for Enterprise Storage”. In: *FAST’08*. San Jose, USA, Feb. 2008.

[56] Michael Nelson, Beng-Hong Lim, and Greg Hutchins. “Fast Transparent Migration for Virtual Machines”. In: *ATC’05*. Anaheim, USA, Apr. 2005.

[57] Kyle Nesbit and James Smith. “Data Cache Prefetching Using a Global History Buffer”. In: *Micro’05*. Barcelona, Spain, Nov. 2005.

[58] Balaji Palanisamy et al. “Purlieus: Locality-aware Resource Allocation for MapReduce in a Cloud”. In: *SC’11*. Seattle, USA, Nov. 2011.

[59] Pierre Riteau, Christine Morin, and Thierry Priol. “Shrinker: Improving Live Migration of Virtual Clusters over WANs with Distributed Data Deduplication and Content-Based Addressing”. In: *Euro-Par’11*. Bordeaux, France, Aug. 2011.

[60] Rusty Russell. “virtio: towards a de-facto standard for virtual I/O devices”. In: *SIGOPS Oper. Syst. Rev.* 42 (5 2008).

[61] Jeffrey Shafer. “I/O virtualization bottlenecks in cloud computing today”. In: *WIOV'10*. CA, USA, Mar. 2010.

[62] Prateek Sharma and Purushottam Kulkarni. “Singleton: System-wide Page Deduplication in Virtual Environments”. In: *HPDC'12*. Delft, Netherlands, June 2012.

[63] Ryan Shea, Haiyang Wang, and Jiangchuan Liu. “Power Consumption of Virtual Machines with Network Transactions: Measurement and Improvements”. In: *INFOCOM'14*. Toronto, Canada, Apr. 2014.

[64] Jason Sonnek et al. “Starling: Minimizing Communication Overhead in Virtualized Computing Platforms Using Decentralized Afnity-Aware Migration”. In: *ICPP'10*. San Diego, USA, Sept. 2010.

[65] T. Sterling et al. “BEOWULF: A parallel workstation for scientific computation”. In: *ICPP'1995*. Urbana-Champain, USA, Aug. 1995.

[66] MECHTHILD STOER and FRANK WAGNER. “A simple min-cut algorithm”. In: *Journal of the ACM* 44 (4 1997), pp. 585–591.

[67] Prateek Tandon, Michael J. Cafarella, and Thomas F. Wenisch. “Minimizing Remote Accesses in MapReduce Clusters”. In: *IPDPSW'13*. Cambridge, USA, May 2013.

[68] Cheng-Chun Tu et al. “A Comprehensive Implementation and Evaluation of Direct Interrupt Delivery”. In: *VEE'15*. Istanbul, Turkey, Mar. 2015.

[69] Ahsen J. Uppal, Ron C. Chiang, and H. Howie Huang. “Flashy Prefetching for High-Performance Flash Drives”. In: *MSST'12*. San Diego, USA, Apr. 2012.

[70] VMware. *Performance of vSphere Flash Read Cache in VMware vSphere 5.5*. 2013. URL: http://www.vmware.com/files/pdf/techpaper/vfrc-perf-vsphere55.pdf.

[71] VMware. *What's New in VMware Virtual SAN*. 2014. URL: http://www.vmware.com/files/pdf/products/vsan.

[72] John Wilkes and Charles Reiss. *Google Cluster Data*. 2014. URL: https://code.google.com/p/googleclusterdata/wiki/ClusterData2011_2.

[73] Timothy Wood et al. "CloudNet: Dynamic Pooling of Cloud Resources by Live WAN Migration of Virtual Machines". In: *VEE'11*. Newport Beach, USA, Mar. 2011.

[74] J. Zebchuk et al. "RECAP: A region-based cure for the common cold (cache)". In: *HPCA'13*. Shenzhen, China, Feb. 2013.

[75] Yiying Zhang et al. "Warming up Storage-Level Caches with Bonfire". In: *FAST'13*. San Jose, USA, Feb. 2013.

[76] Jie Zheng, T. S. Eugene Ng, and Kunwadee Sripanidkulcha. "Workload-Aware Live Storage Migration for Clouds". In: *VEE'11*. Newport Beach, USA, Mar. 2011.

[77] Jie Zheng et al. "COMMA: Coordinating the Migration of Multi-tier Applications". In: *VEE'14*. Salt Lake City, USA, Mar. 2014.

[78] Jie Zheng et al. *Pacer: Taking the Guesswork Out of Live Migrations in Hybrid Cloud Computing*. Tech. rep. TR13-01. Rice University Technical Report, Jan. 2013.

VITA

Tao Lu was born on January 7, 1986, in Xianning City, Hubei Province, China. He graduated from Xianning No.1 Middle School, Xianning, China in 2005. He received his Bachelor of Science in Computer Science and Technology, his Master of Science in Computer Architecture, both from Huazhong University of Science and Technology (HUST), Wuhan, China in 2009 and 2012, respectively. Tao was the winner of the 2016 ECE Outstanding Graduate Research Assistant Award of VCU. Tao was the winner of the Best Student Paper Award of the 9th IEEE International Conference on Networking, Architecture, and Storage (NAS 2014).